Universidad Nacional de Colombia

Master thesis

Static Analysis of Python Programs using Abstract Interpretation: An Application to Tensor Shape Analysis

Author:
Elkin Alejandro Cruz Camacho

Supervisor:
PhD. Felipe Restrepo Calle

Co-supervisor:
PhD. Fabio Augusto González

A thesis submitted in fulfillment of the requirements
for the degree of Masters Systems and Computer Engineering

in the

PLaS - Mindlab
Departamento de Ingeniería de Sistemas e Industrial

8th March 2019

“*The truth is that everyone is bored, and devotes himself to cultivating habits.*”

Albert Camus, The Plague

“*Perhaps I’m old and tired, but I think that the chances of finding out what’s actually going on are so absurdly remote that the only thing to do is to say, "Hang the sense of it," and keep yourself busy.*”

Douglas Adams, The Hitchhiker’s Guide to the Galaxy

Resumen

Abstract

Tensors are, an extension of arrays, widely used in a variety of programming tasks. Tensors are the building block of many modern machine learning frameworks and are fundamental in the definition of deep learning models. Linters are indispensable tools for today's developers, as they help the developers to check code before executing it. Despite the popularity of tensors, linters for Python that check and flag code with tensors are nonexistent. Given the tremendous amount of work done in Python with (and without) tensors, it is quite baffling that little work has been done in this regard. Abstract Interpretation is a methodology/framework for statically analysing code. The idea of Abstract Interpretation is to soundly overapproximate the result of running a piece of code over all possible inputs of the program. A sound overapproximation ensures that the Abstract Interpreter will never omit a true negative, i.e. if a piece of code is not flagged by the Abstract Interpreter, then it can be safely assumed that the code will not fail. The Abstract Interpreter can be modified so that it only outputs true positives, although losing soundness, i.e. the interpreter can flag which parts of the code are going to fail no matter how the code is run.

In this work, we formalise a subset of Python with emphasis on tensor operations. Our formal Python semantics are based on The Python Language Reference. An Abstract Interpreter was defined and implemented from the Python formalisation. We show how each part of the Abstract Interpreter was built: the Abstract Domains defined, and the inferred formal semantics. We present the structure of Pytropos, the Abstract Interpreter implemeted. Pytropos is able to check NumPy array operations taking into account broadcasting and complex NumPy functions as `array` and `dot`. We constructed 80 unit test cases checking the capabilities of Pytropos and 50 property test cases checking compliance with the official Python implementation. We show what and how many bugs the Abstract Interpreter was able to find.

Acknowledgements

Contents

Prelude

Dear reader, thank you for picking this thesis to read. Reading a thesis is often an annoying *task*, and it is also a very difficult task to do as well. Despite what my advisors tell me, I like to write in a style slightly different from the style seen often in science publications. I ask you, dear reader, to tolerate my style of writing, I try to write the text as if I were explaining all the stuff to my past-self (one and a half years in the past). You will find many overexplanatory sentences, paragraphs and complete subchapters, with examples and other uncommon customs. My intention is, after all, to be a guide for my past-self to come where I have arrived, but faster.

Chapter 1

Introduction

1.1 Motivation

Dynamically typed programming languages like Python and Ruby have been rising in popularity over the last two decades. Their simple syntax, ease of use, out-of-the-box REPLs[1], dynamic type systems, and the great number of libraries they come bundled up with have been all factors to their rise. Indeed, being able to write, execute and see results inmediately makes for very speedy development cycles, especially at the initial stages of development. However, their simple syntax and dynamic typing makes them particularly difficult to analyse as a language is easier to analyse as the more rigid it is. The more rigid or restricted a language is the easier to analyse it is. Most efforts on Static Analysis for Python have been focused on assuming some restrictions on the language. For example, MyPy (Lehtosalo et al., 2016) assumes as static the type of variables.

Dynamically typed programming languages only check for type mismatches at runtime, i.e. errors between variable types get only detected when the code is run. For example, consider the following piece of Python code:

```python
myvar = "Some text in a string"
other = 4.1
result = "oh my" + 2  # This will fail
print(result + other, myvar)
```

The code is syntactically correct but it fails to execute. Python will interpret the first two lines and will be stuck on the third, where it will throw an exception because strings and numbers cannot be added.

A Static Type Analysis tool, such as MyPy (Lehtosalo et al., 2016), is able to detect type mismatches without ever executing the code. Mypy has the ability to infer the type of variables, e.g. from the previous example, it detects that `other` is of type `float` and `myvar` of type `str`.

Python dynamic nature makes Python extraordinarily hard to statically analyse. In fact, due to Python introspection capabilities, it is impossible to ever warranty anything about a piece of Python code without assuming some basic behaviour by the interpreter. Mypy tackles the undecidability problem of Python dynamic nature by restricting the number of valid Python programs to those that conform to a static type system[2].

Even with some assumptions in place, Mypy's inference ability only goes so far as to infer the type of variables but not the type of functions. Mypy allows, optionally, to add type annotations to the code to

[1]Read-eval-print loop

[2]I would like to express my inmense gratitude to the MyPy team for writing such an amazing tool. I do not know how many hours they saved me from endless frustration at debugging. What I like more about MyPy is that writing code with it feels just remarkably similar to writing code in language with a stronger static type system (e.g. Haskell).

either variables or functions. Type annotations allow more restrictive and precise types, which help MyPy to catch more potential bugs.

However useful Mypy is, its assumptions limit it from a whole chunk of valid and correct Python code[3]. There are many scenarios where even with type annotations Mypy cannot detect a bug in the code. For example, consider the following piece of python code:

```python
import numpy as np
x = np.array( [[1,2,3], [4,5,6]] ) # shape (2,3)
y = np.array( [[7], [0], [2], [1]] ) # shape (4,1)
z = np.dot( x, y ) # trying to apply dot product
```

We are using in this example a well known Python library, NumPy (Oliphant, 2006), but it could have been TensorFlow (Abadi et al., 2016) or any other library with support for tensors and tensor operators. Tensors are a generalization of vectors and arrays. A vector of size `n`, or array of shape `(n,)`, contains exactly n elements; a matrix of size $n \times m$, alternatively, a matrix of shape `(n, m)`, is a structure that holds $n \cdot m$ elements; and, a tensor is a structure that can have any integer tuple as its shape, e.g. `(2, 5, 6)` which could hold $2 \cdot 5 \cdot 6 = 60$ elements.

The example above will fail to run completely as it hits an erroneous condition at the last line. Two arrays/tensors are created, `x` and `y` with shapes `(2, 3)` and `(4, 1)`, respectively, then a matrix multiplication with both tensors tries to be performed, but it fails because the shapes of the two matrices are incompatible ($3 \neq 4$). The call to `np.dot` fails, but we won't know this until we execute the code. To summarise, the code above type checks in MyPy even though it fails to run.

If it were possible to add the shape of the tensors as part of the type definitions, we could potentially type check for mismatches of tensor's shapes. Unfortunately, there does not seem to be a straightforward way to add the shape of tensors to types and check them in MyPy.

Tensors allow writing computations more concisely. Tensor operations are also highly optimisable and can be run in parallel. With the advent of Deep Learning in every possible realm in which it can be applied, tensors have become more of a central figure in developers toolkit. Tensor operations fail if the tensors are not of the right shape. To statically analyse if operations between tensors are valid, one needs to calculate the shape of tensors which requires to statically analyse the value of variables in the language (also called, Static Value Analysis).

1.2 Problem Definition

Tensors are becoming an essential abstraction in the toolkit of developers. Tools tailored to work with or based on tensors have been popping out in recent years. It is therefore imperative to develop analysis tools to verify and flag potential bugs. A Static Value Analysis tool focused on tensor shape analysis could aid developers in their work, as it could be able to flag potential bugs related to the shapes of tensors as the developer writes code.

All of the theoretical approaches that tackle the problem focus on type checking tensors and tensor operations. Griffioen (2015), Slepak, Shivers, and Manolios (2014), and Rink (2018) define a type system extended with tensors and tensor operations. The principal idea of these type systems is to encode the

[3]For example, there is no way (currently) to make MyPy happy about the function `def addtwice(x, y): return x + y + y; addtwice(3,2); addtwice('a','b')`. If no type annotations is given, MyPy alerts the user on the usage of non-typed functions. But no type can satisfy the assertion because of the "polymorphisity" of the `+` operation.

restrictions that the operations on tensors require, a tensor operation is valid if the restriction checks with the tensors.

Practical attempts have been mainly focused on languages with strong type systems, and recently some in Python (Fromherz, Ouadjaout, & Miné, 2018; Monat, 2018). Chen (2017) (in Scala) and Eaton (2006) (in Haskell) have tried to annotate tensors' types with their shapes, leaving the compiler's type check inference system to check the shapes. At the time Eaton (2006) proposed his methodology to extend types with additional data, GHC (Glasgow Haskell Compiler), the by default Haskell compiler, didn't have many capabilities to work with data at the type level, resulting in some rather complicated code to read and write. A recent approach to type check tensors in Haskell has been shown in a library written by Cruz-Camacho and Bowen (2018), which uses the updated Dependent Type System of Haskell to code and enforce type shapes.

Abstract Interpretation is a framework for static analysis. The main idea of Abstract Interpretation is to overapproximate the result of computing a piece of code. Any piece of code can be run in an Abstract Interpreter, an interpreter build from the semantics defined by Abstract Interpretation, in a finite amount of time. To warranty that the Abstract Interpreter runs in a finite amount of time, Abstract Interpretation defines a series of rules that force to overapproximate the result of executing operations by the program. Abstract Interpretation is especially useful for Static Value Analysis as the rules to derive the semantics of an Abstract Interpreter from the formal semantics of the language are very well studied, formalised and explicit (Cousot & Cousot, 1977).

Writing an Abstract Interpreter for Python requires the formal semantics of Python as a starting point. Unfortunately, Python does not have any official formal semantics defined. Some attempts to define a formal semantics for Python have been done (Politz et al., 2013; Fromherz et al., 2018; Guth, 2013; Ranson, Hamilton, Fong, Hamilton, & Fong, 2008) but none of them take into account type annotations.

Problem: Building an Abstract Interpreter for Python to Statically Value-Analyse tensor operations.

Assumptions and Scope

Very little can be asserted from a piece of code in a programming language like Python without making any assumptions about the environment. For example, consider the following piece of code:

```python
# runme.py
a = int('6') + 2
print(7)
assert a == 8
```

One may be tempted to say that the code above never fails, but it would be mistaken. Thanks to Python's introspection capabilities, the code above could do just about anything. A piece of Python code can be run directly in the interpreter or executed by an `exec` or `eval` statement. The functions `int` and `print` can be redefined to compute anything we want. For example, consider the following piece of code in that calls `runme.py`:

```python
runme = open('runme.py').read()
fakeinit = """
def int(val):
  return 3.0
def print(val):
  global a
  a = val
```

```
"""
exec(fakeinit + runme, {})  # Throws an exception
```

From now on, we will assume that any piece of code written in Python will be called from the interpreter directly without changing the behaviour of any builtin function or variable (this includes the behaviour of variables and functions from non-builtin libraries as NumPy), i.e. we assume that the code will never be run by an `exec` call but always `imported` or interpreted directly by `python file-code.py`.

As a developer, I can say that I like when I see a piece of code flag wrong and IT IS really WRONG, but I despise when the opposite happens. It is very annoying to have a tool flag every sentence you write as wrong even though the code works just fine. We will assume from here onwards that we want the Tensor Analysis to flag only errors that are going to happen if the code is run, i.e. we want only true positives not maybe positives (as a lack of a better word).

Although, TensorFlow (Abadi et al., 2016), PyTorch (Paszke, Gross, Chintala, & Chanan, 2017) and other libraries would benefit from the development of a tool targeted to check them, I chose to check NumPy's tensor operations. NumPy is the standard library for computing with tensors and it is used as backend on most projects.

Implementing an Abstract Interpreter, or any Static Analysis, for a language like Python is a considerable undertaking, mainly because of the breadth of characteristics a mature Programming Language like Python has. Thus, we will centre on just a couple of Python core characteristics and will leave others for future work. The characteristics explored in the current document are:

- Builtin primitive variables: `int`, `float`, `bool`, `None`, `list`, `tuple`
- Primitive functions: `int`, `print`, `input`, ...
- Boolean and numeric operations: +, -, *, /, %, **, <<, >>, //, <, $\leqslant$, > and $\geqslant$
- `if` and `while` staments
- Import statement (limited only to the `numpy` library)
- NumPy arrays and some of its methods/functions to work with NumPy arrays (including `dot`, `zeros`, `shape`, and numeric operations with broadcasting)

Consider the following piece of code:

```
if someval:
  i = 3
else:
  i = "ntr"
print(i)
```

The variable `i` can be either an `int` or a `str`, thus the type of `i` should be `Union[int,str]`. If the type of a variable is an `Union` type then we need to build an Abstract Interpreter (actually, an Abstract Domain) aware of unions. Building an Abstract Interpreter aware of `Union` types is not considered part of this work, therefore if a variable holds more than one value, e.g. it is either a `5` or `2.3`, then the variable's type will be `Any` and not `Union`. `Any` and `Union` types are defined in the `typing` library (Guido van Rossum, Jukka Lehtosalo, & Łukasz Langa, 2014), an implementation of Gradual Typing[4].

Gradual Typing restricts the number of valid programs to those that type check. A variable in Gradual Typing is of a specific type and it never changes, i.e. if a variable's type is set to be `int` then it can only contain `int`s. `Union` types are meant to indicate that a variable may take more than one value at any point in time, which would mean that any variable with a `Union` type would automatically become an `Any` in this work, but that is not the truth. Consider the following piece of code:

[4]Gradual Typing is a kind of Static Type Analysis targeted to Dynamically Typed Languages like Python

```
i = 3
i = "ntr"
i += "ueor"
print(i)  # the type of `i` is `str`
```

Under Gradual Typing, the type of `i` is `Union[int,str]`, but we know that the variable `i` holds only one type at **all** times, i.e. `i` never holds two or more values as in the previous example. An Abstract Interpreter would run each line sequentially, and it would never find a that the value of `i` is both `int` and `str` at the same time, i.e. the variable is never considered to be of type `Any` but it will have type `int` and then type `str` as the code is run. In this regard, an Abstract Interpreter can give a better approximation of what the types of a variable are at any point in time, while Gradual Typing considers all possible types a variable may have at any point in time.

To recapitulate, the assumptions and scope of the thesis are:

- The behaviour of builtins and imported variables and functions is always the same and is determined by the Python reference manual or the library's author, e.g. `input` is a function that returns a `str`.
- The user will only be reported errors that will happen but no errors that *may* happen (This means, the resulting tool is not a verification tool).
- The Static Value Analysis centres around checking operations from the NumPy library.
- Only a selected subset of Python characteristics and functions are explored.
- A variable can hold only one type of value at the time. If a variable is set to hold values of different types at the same time, then its value will be `Any`.

1.3 Objectives

1.3.1 General objective

To design and implement a strategy for statically analyse tensor shapes in Python to support early detection of potential bugs before execution.

1.3.2 Specific objectives

1. To design and implement an abstract interpreter for Python.
2. To design and implement a strategy that uses the abstract interpreter to analyse the shapes of tensors for Python code.
3. To implement a tool that flags the bugs inferred by the abstract interprete.
4. To empirically evaluate the developed static analysis in a set of representative test cases.

1.4 Contributions

The contributions of this work are the following:

- Formalised a subset of Python using a methodology similar to (Fromherz et al., 2018). The formalisation includes a subset of the Python syntax, an AST representation of the syntax more malleable to work with, and the semantics of the subset of the Python Language.
- Defined an Abstract Domain for Variables in Python.
- Defined an aliasing-aware Abstract Domain for the state of a Python program.
- Derived semantics from the Abstract Domain, i.e. an Abstract Interpreter for Python.
- A working open source implementation of the Abstract Interpreter nicknamed Pytropos.

1.5 Thesis Structure

The following parts of this document are as follows:

- Chapter 2 explores some background material on Dynamically typed and Statically typed languages, Tensors and why are they important, the Numpy library, and Abstract Interpretation,
- Chapter 4 presents some of the gory details of the implementation of the Abstract Interpreter,
- Chapter 5 focuses on the tests made to ensure that the implementation follows the Abstract Interpreter, to showcase the utility of Pytropos to detect errors when using NumPy arrays, and to point some of the areas of improvement of the tool,
- Chapter 6 presents some related work on Abstract Interpretation for Python, Static Analysis of Tensor shapes and related libraries Static Analysis tools, and finally,
- Chapter 7 sums up the work done and future directions to it.
- Appendix A is divided into three parts: the definition of the syntax and semantics of a subset of Python, the definition of an Abstract Interpreter for Python, and finally some other ways in which Statically Analyse the shape of tensors for Python,

Chapter 2

Background

In this chapter, we explore a couple of motivational topics and ideas necessary on the development of the Static Value Analysis proposed in Chapter A.

2.1 Dynamic and Static Analysis

Static Analysis and Dynamic Analysis are two disciplines whose ideal is to ensure some property holds on a piece of code. Static Analysis checks the code without the need to run the code, while Dynamic Analysis checks the code as it is executed.

Static Analysis is often associated with compiled Programming Languages. Compiled Programming Languages require to know how to precisely translate the operations of the language into machine instructions. Compiled Programming Languages make use of a variety of Static Analyses to find out the suitable machine instructions for a piece of code.

Interpreted Programming Languages, on the other hand, find out which machine instructions to use for a given operation as they encounter the operations, thus they make heavy use of Dynamic Analyses. Notice that the distinction between Compiled Programming Languages and Interpreted Programming Languages does not stem from their use of Static and Dynamic Analyses but how they process the input files. Compiled Programming Languages read the totality of the input files and translate the operations into machine instructions, while Interpreted Programming Languages do not need to read the whole file before starting to execute the instructions in them.

Static and Dynamic Typing are two categories of the same kind of Analysis, Type Analysis (Pierce & Benjamin, 2002). Type Analysis tells compilers and interpreters the type of a variable. Both Static and Dynamic Typing based Programming Languages have strong and weak points: Compiled Languages hold stronger warranties for the resulting binary as they restrict the set of valid programs to those that "type check", and Interpreted Languages allow the programmer to forgo the usually expensive step of compiling code.

Dynamically Typed, Interpreted Programming Languages (Dynamically Typed Languages) are excellent tools for prototyping as they let the developer test their code without waiting for it to compile. However, writing big pieces of software in a Dynamically Typed Language is often a challenging task as the type warranties are not as strong as in compiled programming languages.

Developers are more prone to make a mistake that will only appear once the code is executed in a Dynamically Typed Languages opposed to Statically Typed (Compiled) Languages. Dynamic Typing is not ideal for code that needs to warranty a high degree of reliability. Reliability defined as the certainty that the code will never fail due to type mismatches.

Traditional Static Typing can be applied to Dynamically Typed Languages to acquire the same level of confidence of Statically Typed Languages. Unfortunately, the restrictions Static Typing induces in the code often make them to much of a hassle for developers of Dynamic Typed Languages.

Several proposals to make Dynamic Type Systems more robust have appeared in recent years. Some notable proposals and implementations include:

- Gradual types for python, Mypy (Lehtosalo et al., 2016),
- Gradual types for Javascript, Typescript (Bierman, Abadi, & Torgersen, 2014; Hejlsberg, 2012) and Flow (Chaudhuri, 2016),
- Gradual types for ruby, using the library rubydust (An, Chaudhuri, Foster, & Hicks, 2011), and
- Refinement types for Ruby (Kazerounian, Vazou, Bourgerie, Foster, & Torlak, 2017).

Gradual Typing (Siek & Taha, 2006) was proposed as a way to bridge the gap between Static and Dynamic Typing. The main idea of Gradual Typing is to enforce Static Typing only to those parts of the code that the developer cares about.

Gradual Typing adds a new type value to the Type System, `?` (named *Any*). In Gradual Typing all variables are of type `?` unless the user annotates them with a more precise type like `int` or `float`. Type annotations are optional in Gradual Typing, as opposed to Static Typing where the type of all variables must be known (either by annotations or inference). Strictly speaking, Gradual Typing is a Static Typing Algorithm that simulates Dynamic Typing by extending the typing rules with the type `?`. Operating with `?` means that we have no idea of the type of the variable we are working with, exactly what happens on Dynamic Typing.

Refinement Types (Rushby, Owre, & Shankar, 1998) are mainly used for verification. The idea of Refinement Types is to make sure a piece of code follows a formal specification. Formal specifications are written in logic and are translated, usually, into propositional formulae to be checked by SAT or SMT Solvers.

2.2 Python Type Annotations

Since Python 3.5 (Guido van Rossum et al., 2014) functions inputs and output can be annotated with types. Since Python 3.6 (Ryan Gonzalez, Philip House, Ivan Levkivskyi, Lisa Roach, & Guido van Rossum, 2016) variables can be type annotated too. Type annotations do no modify the type of the variable they are attached to. Their purpose is only to provide a way to annotate the type the user believes the variable has. Type annotations are used by external libraries as MyPy (Lehtosalo et al., 2016) and Enforce [1] to check for correctness of the code. The Python `typing` library offers a builtin arange of variables to type annotate variables. For an in-depth explanation on the goal of Type Annotations and how are they used in Python refer to (Guido van Rossum & Ivan Levkivskyi, 2014).

2.3 Machine Learning and Python

Machine Learning has been on the rise for the last couple of years since Deep Learning (DL) exploded in popularity. DL has been able to outperform the state of the art in many areas in computer science. DL research consists of finding new models that perform a certain task as well as possible (sometimes even better than humans).

A trained DL model represents the work of many hundreds of hours (often thousand or more) of computing and coding time. Hundreds of DL models have been released to the public by research institutes, the academia, and the industry alike. Given the high amount of competition, it is fundamental for DL practitioners to be able to test many DL models as fast as possible. Trying to find a better model before than others is a fierce fight that drains many hours of laborious thinking, coding and debugging.

[1]GitHub project accessible on https://github.com/RussBaz/enforce

Nowadays, most papers on Machine Learning (ML) and Deep Learning (DL) use Python to define their models. In fact, most DL libraries have an interface to Python. Some of the many Python libraries at disposition for programmers are: TensorFlow (Abadi et al., 2016), PyTorch (Paszke et al., 2017), (the now defunct) Theano (Bergstra et al., 2011), and pycaffe (Jia et al., 2014, as part of the Caffe framework).

Python has become just the right tool for prototyping models, and therefore, Python (and other Dynamic Typed Languages like Lua) has been the primordial playground for DL enthusiasts in the last couple of years.

2.4 Tensors

Array computation is a convenient abstraction to write code that requires the manipulation blocks of variables. The NumPy library presents an example of what can be done with it. Notice how the two pieces of code below perform the same operation ("normalization" of the values in the matrix) but NumPy does it in just fewer lines:

```
import numpy as np
A = np.array([
  [1., 2., 4.],
  [6., 3., 2.],
  [6., 9., 5.],
])

A /= np.max(A)
```

```
A = [
  [1., 2., 4.],
  [6., 3., 2.],
  [6., 9., 5.],
]

m = max(map(max, A))
for i in range(3):
  for j in range(3):
    A[i][j] /= m
```

Multidimensional arrays, arrays which are indexed by not one but two integers, are often called tensors. We take in this work the same approach of calling all–arrays, matrices and tensors–the same way, tensors. If a tensor has one dimension it is equivalent to an array, and if it has two dimensions it is equivalent to a matrix.

Operating with tensors is often syntactically simpler than using looping structures to operate with blocks of memory, i.e. writing `a+b`, where `a` and `b` are tensors is simpler than `[a[i]+b[i] for i in range(len(a))]`. Besides, `a+b` even works for tensors with different (but compatible) shapes.

2.5 NumPy: Library for tensor computation

The primary purpose of this work is to build a Static Value Analysis to check for tensors and tensors operations. NumPy (Oliphant, 2006) is a very widely used library to operate with tensors and arrays, in fact, many other libraries are built on top of NumPy.

NumPy tensors are called arrays. An array can be created out of almost anything, almost any type of value can be interpreted into an array by NumPy[2]. For example, all of the following variables hold a valid NumPy array:

```
a = np.array("some text")       # array of chars (8-bit numbers)
b = np.array([[1, 2], [2, 4]])  # array of ints of shape (2, 2)
c = np.array([[1, 2], [2]])     # array of objects of shape (2,)
d = np.array([(1, 2), [2, 5]])  # array of ints of shape (2, 2)
e = np.array([[1, 2], {2, 5}])     # array of objects of shape (2,)
```

The shape of a NumPy array can be changed without changing its contents:

```
a = np.arange(50)
a = a.reshape((1,-1,2,5))
```

Reshaping preserves the number of elements in an array. If an `-1` is found, it will be taken as a wild card and the missing dimension will be calculated to keep the original shape.

Numpy has this little, nice trick to handle operations that would require reshaping or copying an array when operating with two arrays with different, but similar, shapes. The trick is called broadcasting, and can be better understood with an example:

```
a = np.zeros((10, 3, 4)) + 2
b = np.arange(3).reshape((3, 1))
c = a * b
```

Notice that the code is valid and evaluates in Python. The shape of `a` is `(10, 3, 4)` because adding an array of any shape to an `int` gives us back an array with the same shape, i.e. we have added a scalar to every element of a tensor. `b` has a shape of `(3, 1)` and the shape of `c` is `(10, 3, 4)`. Broadcasting "generalises" the rule of operating with a scalar (as it is done in the first line). The rule works by extending with `1`'s the left side of the smaller shape until both are equal, and then checking if the dimensions of both shapes are compatible. Two dimensions are compatible if are the same or one of them is `1`. For a deeper explanation on broadcasting take a look at https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html.

2.6 Abstract Interpretation

In this section, a rather informal description of Abstract Interpretation is given. For an in-depth explanation of Abstract Interpretation refer to (Nielson, Nielson, & Hankin, 2015, Chapters 1 and 4; Cousot & Cousot, 1977; Nipkow & Klein, 2014).

2.6.1 Informal Introduction

Static Analysis includes a broad assortment of techniques with the purpose of verifying some property in the code. It may be trying to prove some invariant in the code, like there is never a segmentation fault, the code does what it says it does even when ran in concurrently, or the codes terminates (given some restrictions (Urban, 2015)).

[2]Notice that this represents a challenge for Static Type Analysis as it requeries the type of a variable to depend on the contents of the variable. Fortunately calculating the contents of a variable is what Abstract Interpretation does. An Abstract Interpreter allows us to implement/replicate the whole semantics of a library in a similar way at how it is defined in the original library

Abstract Interpretation was conceived (Cousot & Cousot, 1977) as a way to approximate the result of all possible executions of a computation. Specifically, Abstract Interpretation computes an **over**approximation of some property of the program. To explain what an overapproximation is consider the following piece of code:

```
a = int(input())
a = abs(a)
b = 2 * a
```

Assuming the user inputs a valid number and the code runs as intended, we know at the end of the computation that: `a` holds a non-negative integer, and `b` holds a non-negative integer multiple of 2. An overapproximation of this computation is saying that `a` and `b` are both non-negative integers (notice that we do not say that `b` is even).

There is an inherent tradeoff between the precision of an overapproximation, and the amount of storage and processing the overapproximation requires. The most precise method is to store in a set all the possible values a computation may take but it is often impossible to do. The example above asks us to store an infinite set of numbers, which is impossible.

What is very interesting about Abstract Interpretation is that it is based on solid theory. If one applies it correctly, a Static Analysis based on Abstract Interpretation is warrantied to be sound, i.e. it can warranty there will never be a miss or false negatives. Although given the nature of overapproximating, the Static Analysis may produce a load of false positives. The amount of false positives depends on the specific overapproximation used.

How is the overapproximation calculated and which tools do we have at our disposal to compute it? The following subsections present briefly the important concepts behind Abstract Interpretation. For an in-depth explanation of Abstract Interpretation refer to (Nielson et al., 2015, Chapters 1 and 4; Cousot & Cousot, 1977; Nipkow & Klein, 2014).

2.6.2 Ingredients for an Abstract Interpreter

Before we start we need some groundwork. First of all, as always in Programming Languages, we need a language to analyse together with its formal semantics. We also need a property we want to analyse.

Specific for this work, the property to analyse will be the value of variables throughout the execution of the program. i.e. Value Analysis. Abstract Interpretation can be used to analyse any other property of programs, some examples include: memory consumption, functional properties, thread safety, computation traces, and termination.

Before we can analyse a piece of code, we require to know the language to analyse. For Value Analysis we require the following description of the language:

- The syntax of the language (how to write things),
- The values the language handles (for example, integers, booleans and floats),
- The state of a program (how to group the values and their names), and
- The concrete semantics of the language (the rules that tell us how to evaluate the code)

Given these we can define:

- An Abstract Domain for the values in the language (the overapproximation used to represent values),
- An Abstract Domain for the state of the program (overapproximates the state of the program), and
- The abstract semantics induced from the Abstract Domain and the concrete semantics.

2.6.3 Value Abstract Domain

Our goal with Abstract Interpretation is to overapproximate the result of running a program on all possible inputs. Remember that a variable holds a single value at the time, so a naïve approach to run over all possible inputs is to extend what a variable can hold.

Suppose a variable can hold a set of values, not a single value, and we extend the semantics of a language to operate over all combinations of values any time an operation is made between two variables. If we do this, we will be able to calculate all possible states a program may arrive. For example, consider:

```
cond1 = bool(input())  # Stdin input
cond2 = bool(input())

if cond1:
  a = 0
  b = 20
elif cond2:
  a = 2
  b = 6
else:
  a = 1
  b = 9
```

After running the code, we know that `a` $\in \{0, 1, 2\}$ and `b` $\in \{6, 9, 20\}$. Now, if we want to compute `a+b` we must compute all possible results of computing with the values from `a` and `b`, which results in `a+b` $\in \{6, 9, 20, 7, 10, 21, 8, 11, 22\}$. Notice how the result of calculating `a+b` overapproximates the real values `a+b` can take, i.e. `a` and `b` are assumed to be independent of each other so we get way more possible values for `a+b` than can actually be computed (we overapproximate).

It is clear that computing with sets of values is expensive, even impossible at times, e.g. the set $\{a \in \mathbb{N} | a \leq 0\}$ cannot be stored in memory.

What we can do to not compute all possible computations is to use an abstraction that lets us overapproximate the values contained in the sets. For example, if we use interval arithmetic to approximate the sets we get:

`a` $\in [0, 2]$, `b` $\in [6, 20]$, and `a+b` $\in [6, 22]$

Which can be easily stored and manipulated (we only require two numbers to encode an interval). Notice how any abstraction will make us lose some precision. This is the inherent tradeoff of computing using Abstract Interpretation, we lose precision.

Notice that overapproximating the value of variables can lead us often to think that a piece of code may be erroneous even if it does not. Consider:

```
b = bool(input())

if b:
  a = 2
  b = -1
else:
  a = 6
  b = 3

c = a / b  # c is either -2 or 2
```

It is simple for us to run this example in all possible inputs, as there are only two, an notice that the code never fails. If `b` is `True`, then we know that `c` is `-2`, and if `b` is `False`, then we know that `c` is `2`. We know that no matter the input, the code will never fail! But when we abstractly interpret the code using intervals as an overapproximation, we get that `a` $\in [2, 6]$ and `b` $\in [-1, 3]$, so when we try to run `a / b` we are alerted that we may be dividing by zero[3].

When we abstractly interpret a piece of code and get an error, the error may not exist.

On the other hand, if we abstractly interpret a piece of code and find no error in it, we can be sure that the code will never fail because we have tested the code on an **over**approximation of all possible values[4]. This property is called *soundness* and it is central to verification and other areas of Static Analysis.

Notice that we do not care about soundness in this work. Our goal is to build a tool that does not overwhelm the common developer but helps them to find *true* bugs! i.e. we do not want false positives just as many true positives as possible.

Fortunately, if the overapproximation we are using allows us to distinguish between single values, e.g. the overapproximation can tell us that `a` has been set to the value `5` after the assignation `a = 5`[5], then it is possible for us to check if two single values are equal or different. If we are able to assert with certainty that two values are equal or different, then we are able to find true positive comparisons (oppossed to maybe `True` comparisons)[6].

The formal definition of an overapproximation is an Abstract Domain. An Abstract Domain is composed of a lattice, a Galois connection between a set of values and the lattice, and widening and narrowing operators.

A lattice is a partially ordered set $(L, \leq)$ with the following properties:

- Every two elements $a, b \in L$ have a lower bound ($u \in L$ such that $u \leq a$ and $u \leq b$). The operation of finding the lower bound between two elements is called **merge** and is denoted by $a \sqcap b$.
- Every two elements $a, b \in L$ have an upper bound ($u \in L$ such that $a \leq u$ and $b \leq u$). The operation of finding the upper bound between two elements is called **join** and is denoted by $a \sqcup b$.
- There is an element bigger than any other ($u \in L$ such that $\forall x \in L.x \leq u$), and it is denoted by $\top$.
- There is an element smaller than any other ($u \in L$ such that $\forall x \in L.u \leq x$), and it is denoted by $\bot$.

Given two lattices, $(L_1, \leq_{L_1}, \sqcap_{L_1}, \sqcup_{L_1}, \top_{L_1}, \bot_{L_1})$ and $(L_2, \leq_{L_2}, \sqcap_{L_2}, \sqcup_{L_2}, \top_{L_2}, \bot_{L_2})$, a Galois connection is a pair of functions (α, γ) such that:

- $\alpha : L_1 \to L_2$. α is called the abstraction function.
- $\gamma : L_2 \to L_1$. γ is called the concretisation function.
- $\forall a \in L_1, b \in L_2 : \alpha(a) \leq_{L_2} b \iff a \leq_{L_1} \gamma b$.

L_1 is often called the Concrete Domain and L_2 is called the Abstract Domain. Because the Concrete Domain is always the same, the set of possible values a variable can have with $\in$ as an order operator, we concentrate our efforts in the Abstract Domains.

Notation: It is customary to use the symbol $\#$ to refer to the functions and semantics associated with the Abstract Domain, and the lack of it as the functions and semantics of the Concrete Domain. For example,

[3] dividing by zero throws an Exception in Python

[4] assuming no builtin value has been changed prior to the execution of the code

[5] The sign overapproximation (or Sign Interval Abstract Domain) has only three posible values: `-`, `0` and `+`, while the interval overapproximation (or Interval Abstract Domain) contains any interval $[a, b]$ with $a, b \in \mathbb{R}$. Therefore, if we use the Interval Abstract Domain we are able to tell when we know a variable has a specific value rather than many. On the other hand it is imposible for use to determine a unique value using Sign Interval Domain other than the case `0`.

[6] Consider the result of `[4,4] == [3,3]` (`False`) and the result of `[3,8] == [2,3]` (it may be `True` or `False`)

if Int is the set of all integers then $Int^{\#}$ is an Abstract Domain for Int, e.g. given the Concrete Domain $(\mathcal{P}(Int), \subseteq, \cap, \cup, Int, \emptyset)$ we define the Abstract Domain $(Int^{\#}, \leq^{\#}, \sqcap^{\#}, \sqcup^{\#}, \top^{\#}, \bot^{\#}, \alpha, \gamma)$, where $\alpha : Int \to Int^{\#}$ and $\gamma : Int^{\#} \to Int$.

Coming back to the intervals example from before. Given the Concrete Domain of integers $(\mathcal{P}(Int), \subseteq, \cap, \cup, Int, \emptyset)$, we can define the Intervals Abstract Domain Int$^{\#}$ as (Int$^{\#}$, $\leq^{\#}, \sqcap^{\#}, \sqcup^{\#}, [-\infty, \infty], \emptyset, \alpha, \gamma)$ where:

- Every element of Int$^{\#}$ is an interval $[a, b]$ with $a, b \in Int \cup \{-\infty, \infty\}$.
- $[a, b] \leq^{\#} [c, d] \iff c \leq a \wedge b \leq d$.
- $[a, b] \sqcap^{\#} [c, d] = [max(a, c), min(b, d)]$.
- $[a, b] \sqcup^{\#} [c, d] = [min(a, c), max(b, d)]$.
- $\alpha(I) = [min(I), max(I)]$ (Remember that I is a set of integers).
- $\gamma([a, b]) = \{i \in Int : a \leq i \leq b\}$.

The goal of an Abstract Domain is to give us a very powerful tool, the Galois Connection. The abstraction function allows us to take any value in our language and transport it to an overapproximation of our choosing, and the concretisation function allows us to take abstract values (overapproximations) and operate with them back in the semantics of our language.

Given, for example, the Intervals Abstract Domain, we can now calculate the result of computing the following piece of code:

```
if _:
  a = 2
else:
  a = 4
a -= 6
```

We know that the value of a in the *then* branch of the of the *if* statement is equal to `[2,2]` and `[4,4]` in the *else* branch. The value of a after the execution of the *if* statement will be `[2,2]` $\cup$ `[4,4]` $=$ `[2,4]`. At the end of the execution the value of a lie inside the interval `[-4,-2]`.

Notice that with a (Value) Abstract Domain, we can overapproximate the value of a single variable but we cannot warranty **termination** of the evaluation of the code. Take for example:

```
a = 4
while not condition(a):
  a += 3
```

Unrolling the loop, we get:

```
a = 4     # a \in [4,4]

  a += 3  # a \in [4,7]
  a += 3  # a \in [4,10]
  a += 3  # a \in [4,13]
  a += 3  # a \in [4,16]
  ...
  # a \in [4, inf]
```

Because we do not know when `condition(a)` will be met, it is impossible for us to know how many times will the `while`'s body be executed, an so we know that a may be either 4, 7, 11, or any other integer. An Abstract Interpreter is an interpreter and therefore will run the body of the while loop forever if we do not do something to terminate it.

Applying a series of operations (the body of the while loop) over and over may never terminate. Thus we have the need a mechanism that allows us to ensure the evaluation of a loop will eventually terminate. Introduce the *widening* operator.

The idea of a widening operator is to make a series of ascending values reach a fixed point in a finite amount of time.

Consider the (increasing) sequence from before[7]:

```
[4,4] [4,7] [4,10] [4,13] [4,16] [4,19] [4,22] [4,25] ...
```

A widening operator is an operator, ϕ, that takes two intervals, a and b, and outputs a new interval, $c = a\phi b$, such that the new interval contains the old intervals $a, b \leq c$. The widening operator must warranty that when applied over and over an increasing sequence gives us a sequence that will, after a finite amount of steps, find a fixed point (stabilizes). Take for example the following widening operator:

$$[a,b]\phi[c,d] = \begin{cases} [min(a,c), \infty], & \text{if } b > 15 \\ [min(a,c), max(b,d)], & \text{otherwise} \end{cases}$$

Applying the operator over our sequence over an over we will get:

```
[4,7] [4,10] [4,13] [4,16] [4,inf] [4,inf] [4,inf] ...
```

And we found a fixed point, namely, `[4,inf]`. We have arrived now at the top of the latter.

Notice how the cutoff of the sequence must be defined beforehand (in our case any interval with a value bigger than `15` defaults to ∞). This is a little bit annoying as one would like for the Abstract Interpreter to work under any condition, but this is the price to pay for termination, we need to set some parameters by hand.

There is another operator called the narrowing operator. Its purpose is to climb down the latter and get a more precise approximation. Notice that not all Abstract Domains require the definition of widening and narrowing operators as not all of them have infinite increasing or decreasing sequences. And it is the case in this work that none of them applies, there are no infinite increasing or decreasing sequences in the Abstract Interpreter defined in this work.

For a deeper discussion on all numerical Abstract Domains available see Miné (2004).

2.6.4 State Abstract Domain

The Abstract Domains presented before lets us analyse one variable at the time. To analyse the state of the program we define an Abstract Domain for the state of the program. How to do this? Well, if we assume that the language allows no aliasing, then we can build the State Abstract Domain as:

```
State: Var -> Val
State#: Var -> Val#
```

[7]Notice that: `[4,4]` $\leq$ `[4,7]` $\leq$ `[4,11]`

If the `State` of the program is defined as a function from `Vars` into their values (`Var → Val`), and `Val#` is the Abstract Domain for values in the languages, then the function `Var → Val#` is an Abstract Domain[8] for the state of the program.

As an example consider:

```
int main() {
  int cond1, cond2, a, b, c;
  std::cin >> cond1 >> cond2;

  if (cond1) {
    a = 0
    b = 20
  } else if (cond2) {
    a = 2
    b = 6
  } else {
    a = 1
    b = 9
  }
  c = a + b;
  return 0;
}
```

Note: This example is written in C++ because we can assume all variables to have a set type, a unique type. In C++, a variable always has a value even if none is given to it. In Python, a variable can have no value, or it can be undefined (as it happens when a variable is deleted).

Suppose that the State Abstract Domain for this piece of code is a function from the set $\{cond1, cond2, a, b, c\}$ to the Interval Abstract Domain. If we evaluate the code line by line we will get:

```
// T: means Top
int main() {
  int cond1, cond2, a, b, c;
  // {'cond': T, 'cond2': T, 'a': T, 'b': T, 'c': T}
  std::cin >> cond1 >> cond2;
  // {'cond': T, 'cond2': T, 'a': T, 'b': T, 'c': T}

  if (cond1) {
    a = 0
    // {'cond': T, 'cond2': T, 'a': 0, 'b': T, 'c': T}
    b = 20
    // {'cond': T, 'cond2': T, 'a': 0, 'b': 20, 'c': T}
  } else {
    if (cond2) {
      a = 2
      // {'cond': T, 'cond2': T, 'a': 2, 'b': T, 'c': T}
      b = 6
```

[8]For a deeper look and a proof on this statement see Nielson et al. (2015), subchapter 4.4.

```
      // {'cond': T, 'cond2': T, 'a': 2, 'b': 6, 'c': T}
    } else {
      a = 1
      // {'cond': T, 'cond2': T, 'a': 1, 'b': T, 'c': T}
      b = 9
      // {'cond': T, 'cond2': T, 'a': 1, 'b': 9, 'c': T}
    }
    // {'cond': T, 'cond2': T, 'a': [1,2], 'b': [6,9], 'c': T}
  }
  // {'cond': T, 'cond2': T, 'a': [0,2], 'b': [6,20], 'c': T}
  c = a + b;
  // {'cond': T, 'cond2': T, 'a': [0,2], 'b': [6,20], 'c': [6,22]}
  return 0;
}
```

Excellent! We have done it! We have the idea of how to build an Abstract Interpreter for a language with only one type of variable, a global scope, and no aliasing.

2.6.5 Abstract Semantics

Now we have our ingredients all ready: We have a language with its concrete semantics and State Abstract Domain. What we have left to do is to define the abstract semantics of the language. The abstract semantics work on the State Abstract Domain, where as the concrete semantics work on the state of the program.

For this, we need the Galois connections defined in the State Abstract Domain, the abstraction α and concretisation γ functions. We can infer the abstract semantics with the following little formula:

$$f^{\#} = \alpha \cdot f \cdot \gamma$$

where f is a function (a rule) in the semantics of the language.

The idea is simple, if f is defined properly defined then we can compose it with the abstraction and concretisation functions and we will get a function $f^{\#}$ which operates over the State Abstract Domain.

This step is important if we want to prove the Abstract Interpreter to be sound and work as expected. For a deeper discussion on how to infer the abstract semantics see Nielson et al. (2015), chapter 4.

2.6.6 Missing Bits

A robust Abstract Interpreter should be able to do more than just what has been said here.

Miné (2004) extends an Abstract Interpreter with the backward assignment. The backward assignment is meant to restrict the possible values a variable may have when it enters an *if* statement. Consider the following piece of C++ (a and b are integers):

```
// {'a': [2,4], 'b': Top}
if (a < 3) {
  // Appling backward assignment. `a<3 and a in [2,4]`
  // {'a': 2, 'b': Top}
  b = a;
  // {'a': 2, 'b': 2}
} else {
  // Appling backward assignment. `a ⩾ 3 and a in [2,4]`
  // {'a': [3,4], 'b': Top}
  b = a - 2;
  // {'a': [3,4], 'b': [1,2]}
}
// {'a': [2,4], 'b': [1,2]}
```

Notice that if we ignore backward assignment the interval for `b` is `[0,4]`. Backward assignment lets us narrow down the overapproximation, which means that we can have a tighter overapproximation and consequently less false positives.

In this work, we make use of only non-relational Abstract Domains. A relational Abstract Domain keeps some of the relationships between variables, they are more expensive than non-relational Abstract Domains but they allow even tighter overapproximations. See more Miné (2004).

Chapter 3

Abstract Interpretation for Python

This chapter explains the formalisation of the proposed Abstract Interpreter. The chapter is divided into three: what are semantics, concrete semantics for Python, and abstract semantics for Value Analysis. This chapter is intended as a guide on why and how was the Abstract Interpreter built. For a deeper look on the details presented in this chapter, please refer to Appendix A.

3.1 Semantics

In Mathematics, the word "semantics" refers to the meaning of sentences in formal languages. The semantics of programming languages consist of what a piece of code means, how to interpret it, and how to analyse it.

classifies "semantics" into three categories: axiomatic semantics, denotational semantics, and operational semantics. Axiomatic semantics are all about describing the behaviour of code with mathematical logic. Denotational semantics goal is to find a mathematical "object" that represents what a program does. The program's meaning is given by the external object. Operational semantics are concerned with rules that tell how to interpret or execute a piece of code. No meaning is given to the parts of the program, but a meaning is given as the program is run.

```
Semantics of Programming Languages
|-- Axiomatic Semantics -- Hoare Logic
|-- Denotational Semantics -- ???
|-- Operational Semantics -- Small-step semantics
                         |-- Big-step semantics
```

Notice that Denotational semantics must not provide a way to determine an implementation. On the other hand, an implementation can be easily build out Operational Semantics. From the many Operational semantics there are, small-step semantics are precisely those that give us the most direct route to see what a program, they are intuitive and every developer could just understand them directly, they define what the program does in a syntax oriented way, is all about applying rules, syntactic rules. It is not a surprise that all efforts into defining formal semantics for Python have been small-step semantics .

In this work, we are no exception. Our purpose is to define the small-step semantics of a portion of Python to show what is our Abstract Interpreter able to do and why.

3.2 Concrete Semantics of Python

Before we define the concrete semantics of Python we need to define how to write a piece of code in Python. Python is a mature and vast language, thus we will take just a portion of it into account. In the figure below, we present the subset of Python that we intend to study in this work.

```
mod = stmt*  -- A program. Starting point

expr = Int(i) for i \in \N | Float(j) for j \in floats
     | True | False | None

     | identifier         -- variable name

     | expr op expr       -- eg, a + 5
     | expr cmpop expr    -- eg, a < 5
     | expr(expr*)        -- Function calling

     | expr.identifier    -- Attribute access
     | expr[expr]         -- Not supported for Numpy Arrays :S
     | [expr*]            -- list
     | (expr*)            -- tuple

stmt = del expr           -- delete expression
      | expr = expr       -- assignment
      | expr op= expr     -- augmented assignment
      | expr: expr = expr -- type annotation
      | while expr: stmt+
      | if expr: stmt+
      | import alias+
      | from identifier import alias+
      | expr              -- An expression can be an statement

op = + | - | * | / | % | ** | << | >> | //
cmpop = < | ⩽ | > | ⩾

alias = identifier | identifier as identifier

identifier = string  -- with some restrictions
```

The details on how to check for the correct number of spaces before expressions to indicate properly when a stament is inside a `while` or an `if` is left unexplained. We do not care about that here.

Let's consider the simple program:

```
b = 2
a = b < 3
```

We can define the following rules:

Notice that to define these rules we make use of two important things, a global scope and a heap. We store the stuff in the heap and point to them with the global scope function. The definition of the small-steps semantics of the language requires us to save the results of execution somewhere.

We call these rules, the small-step semantics of our subset of Python. Actually, they are just a portion of the total rules defined. We don't present all the rules here because it takes too much space. Refer to the appendix to see them in more detail.

With our rules we can "execute" our example:

Defining the small-step semantics on the syntax presented before is a hazzle. Consider the statement delete, this stament is able to not only delete values from the global scope (`del a`), but it can also delete elements from a list (`del ls[2]` deleting the third element of a list `ls`). Notice that it whatever is being deleted by `del` is an expression (in the examples, a and `ls[2]` are both expressions), what should the expression return? (expressions always return stuff). It is simply very hard to know what to return in a case like `ls[2][0].val[1]`, so what we opted to do is exactly what CPython, the official implementation of Python does, that is, extend the syntax to add a little hint of what the purpose of the expression being evaluated will be.

Our new syntax will be:

```
mod = Module(stmt* body)

expr = Int(n)
     | Float(n) | True | False | None
     | Name(identifier, expr_context)
     | BinOp(operator, expr, expr)
     | Compare(expr, cmpop, expr)
     | Call(expr, expr*)
     | Attribute(expr, identifier)
     -- No need for expr_context because no user made objects are allowed yet, thus
     -- modifying attributes is not necessary
     -- | Attribute(expr, identifier, expr_context)
     | Subscript(expr, expr, expr_context)  -- No arbitrary slice allowed yet
     | List(expr*)
     | Tuple(expr*) -- No expr_context for Tuple as `(a, b) = 1, 2` is not supported

stmt = Delete(expr+)
      | Assign(expr, expr)                -- a = 3
      | AugAssign(expr, operator, expr)   -- a += 3
      | AnnAssign(expr, expr, expr)       -- a: int = 3

      | While(expr, stmt+)
      | If(expr, stmt+, stmt*)

      | Import(alias+ names)
      | ImportFrom(identifier, alias+)

      | Expr(expr)

-- Indicates why are we looking up a variable, attribute or subscript
expr_context = Load | Store | Del

operator = Add | Sub | Mult | Div | Mod | Pow | LShift
           | RShift | FloorDiv

cmpop = Lt | LtE | Gt | GtE
```

```
-- import name with optional 'as' alias.
alias = (identifier, identifier?)
```

Notice that this syntax has been directly taken from the CPython implementation, they did in fact find this problem ages ago and it is coded in the language. This new syntax is the result of the parser. The parser takes code written in the first syntax and defines a proper context (Load, Store, or Del) to a couple of expressions. All other expressions are considered Load by default.

Accordingly, our concrete semantics change. First the state of a program is:

and the semantics per se:

Notice the complexity of the type of the functions has increased. Now an expression can not only return a value but three other things, the semantics chose which want to return depending on the value.

Same example as before with the new syntax and semantics:

3.3 Abstract Semantics

In this sections we will intuitively build an abstract domain for the state of the language, and we will define the abstract semantics of the language.

3.3.1 State Abstract Domain

Notice that it is not enough with the primitive values, we need a place where to put them. Whe define the abstract state as

A throughout explanation of how was the whole AD can be seen in the appendix .

Remember the parts of an Abstract Domain, most importantly how to unite two different states of a program.

Now that we have defined our abstract state, we need to define the abstract semantics (just like we defined our concrete semantics based on the state).

3.3.2 The semantics

We present the abstract semantics here, for a little bit more rigourus treatment of them take a look at the appendix.

Defining the abstract semantics is quite straightforward:

To make the abstract interpreter useful, in the implementation we took care of adding checks at any needed place. For this, we log every time the program would die with the concrete semantics now where they do not die because the abstract interpreter returns the wild card value of Top.

Chapter 4

Pytropos (Analysis Implementation)

Pytropos[1] is the Abstract Interpreter implemented in this work. The interpreter follows the rules exposed in the previous chapter.

This chapter is divided into two parts. In the first part, we explore how is Pytropos built, a brief history and some internal details. The second part is a small guide on how to interpret Pytropos output over a couple of different examples.

4.1 The big picture

Pytropos works in a similar way as any other Python interpreter does. It reads code and executes it expression by expression. Its main difference to CPython is that it does not transform code into bytecode to execute but it wraps code before it is transformed into CPython bytecode. The code is wrapped to use a library that implements the semantics of the Abstract Interpreter.

Similar to how Python works, Pytropos can not only "run" individual files, but also offers a REPL (Read-eval-print loop) to check small portions of code.

4.2 Wrapped code + libs + interpreter vs. from scratch interpreter

Ortin, Perez-Schofield, and Redondo (2015) show how to build a Static Type Analysis by rewriting code to operate with the types of values rather than with the values, i.e. the result of executing the code is not some values but some types. For example, consider the following small piece of code:

```
a = 2 + 4
b = a + "yep"  # this will fail
c = a / 2
```

Notice that how even though the piece of code fails to run successfully, we can determine the types of all variables. `a` has type `int`, `b` has no type as it cannot be computed, and `c` has type `float` because the division of `int`s in Python 3 gives us back a `float`.

We can build a library that operates on types (rather than values) and then we can rewrite the piece of code above into:

[1]There is an old trend where programs, companies and anything that could be named was named after some mythological creature from an ancient civilization. Thus, why could not this work also have a proper ancient-based name. Pytropos comes from the merge of Python and Atropos. In the Greek mythology, Atropos was one of the three goddesses of fate who decided on the lifes of humans, she was the goddess of death and the one who cut the thread of life.

```
import typeops as to
a = to.add(int, int)
b = to.add(a, str)
c = to.div(a, int)
```

If the library has properly defined the methods `add` and `div`, we can be sure that the code will run without any error. Notice that the library is able to find type mismatches by embedding checks in the functions `add` and `div`. The library can, effectively, perform Static Analysis on a piece of code.

Pytropos is implemented following the same strategy: a library that operates over abstract values, and a transformation procedure that takes the code and wraps it to use the library. The final step is to run the wrapped code and collect the generated errors[2].

The main advantage of translating code into wrapped code is the reuse of infraestructure. One does not need to write all the infraestructure that an interpreter needs. Pytropos does not implement its own call system, heap management or garbage collection. All of it is managed by the underlying Python implementation where the code is executed.

Nevertheless, this approach has three main disadvantages. First, all operations, function calls, attribute access, subscript access and the whole Python semantics, must be coded into the library and in the transformation function. Second, the place where any operation has occurred must be preserved too, otherwise it is impossible to find where an error has occurred. Finally, all variable access must be wrapped too, because accessing to an undefined variable must stop execution. Consequently, the transformation does not produce a simple, human-readable output. We can run Pytropos with the small code above as input and will get:

```
import pytropos.internals as pt
st = pt.Store()
pt.loadBuiltinFuncs(st)
fn = 'test.py'
st[('a', ((1, 0), fn))] = pt.int(2).add(pt.int(4), pos=((1, 4), fn))
st[('b', ((2, 0), fn))] = st[('a', ((2, 4), fn))].add(pt.str('yep'), pos=((2, 4), fn))
st[('c', ((3, 0), fn))] = st[('a', ((3, 4), fn))].truediv(pt.int(2), pos=((3, 4), fn))
```

Not as nice as the first example.

Pytropos started out from the same idea as Ortin et al. (2015) but it differs on its principal goal. Pytropos goal was not to perform Static Type Analysis but Static Value Analysis. After working for three months on a prototype, it became blatantly clear that trying to wrap the code naïvely did not work very well, i.e. the code was a hacky and not very resilient. The library and transformation needed to be based on a solid theoretical framework.

Enter Abstract Interpretation. Abstract Interpretation offers the ideal framework for Static Value Analysis, it is well understood, with solid theory and extensive work on it has been done for the last four decades.

Ortin et al. (2015) strategy alone may still be a good idea for Static Type Analysis, but it may not work without a framework to glue together the semantics of the languages with those of the analysis. Their legacy to Pytropos is the reuse of Python infrastructure by wrapping the code and not building an interpreter from scratch.

[2]This strategy has been applied on the past for similar purposes, reuse of infrastructure. It was used by Lauko, Ročkai, and Barnat (2018) for symbolic computation of LLVM bytecode.

4.3 Assumptions

Pytropos is limited to work only with Python 3.6 or higher. Pytropos uses variable annotations to allow the user to specify the shape of NumPy arrays when Pytropos is not able to "infer" their value. Variable annotations were introduced on Python 3.6 (Ryan Gonzalez et al., 2016).

The goal of Pytropos is to warn the user when an operation will fail at runtime. It is not a goal for Pytropos to verify the code and prove its correctness (Pytropos is not a tool for verification).

Pytropos goal is not to replace MyPy, flake8, or any other static analysis python tool[3]. Pytropos is meant to be an aid for developers when working with tensors.

Based on that, I present the main assumptions taken into account in the implementation of Pytropos:

- The user wants as little warnings on the code as possible. Pytropos should warn the user for errors it is sure will occur at runtime.
- The user cares only about the shape of tensors and not about the actual values a tensor holds.
- If Pytropos is not able to infer the value of a variable, the user can (optionally) annotate the type/value of the variable. If the annotation is not preciser than the value that Pytropos has already inferred the inferred value will not change.

4.4 Details on the guts

To start with, I do not follow the structure defined in the previous chapter for how elements are saved in memory. I did not explicitly defined a Heap but make use of Python's heap. The main reason to not define a custom Heap is the cost associated to it, especially the definition of a Garbage Collector. The classes `AbstractMutVal` and `Store`, the implementations of `Object#` and `G#`, respectively, do not point to `Addr#`s but they point to `PythonValues` directly (the implementations of `Val#`). The global scope, `Store`, is an object that takes a `str` and returns a `PythonValue`. An `Object#`, `AbstractMutVal`, is an object that has an associated type, operations and can point to any `PythonValue`. In this way, the Pytropos resembles more the graph representation than the classical Heap representation[4].

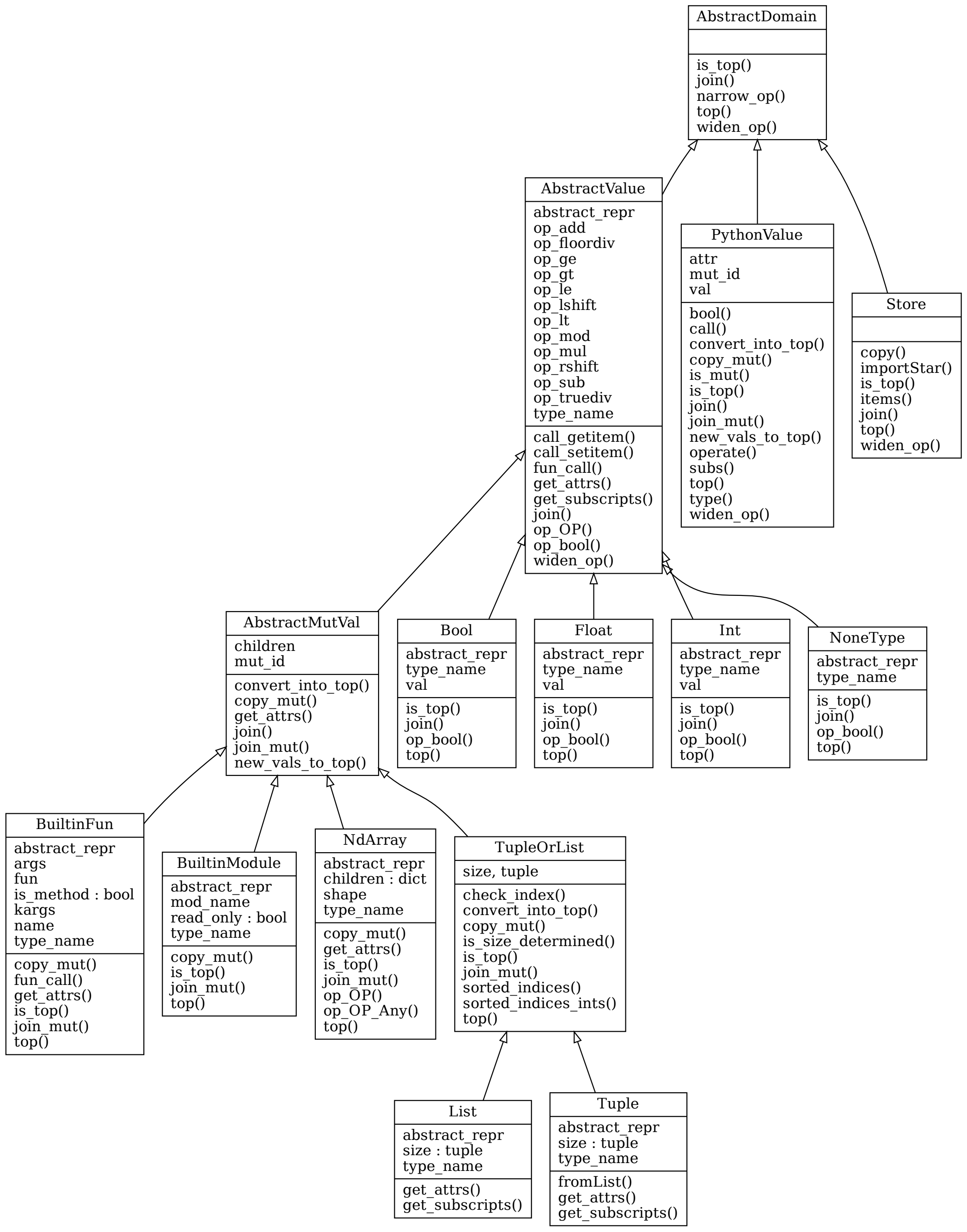
The class `PythonValue` implements `Val#` Abstract Domain. A `PythonValue` is a wrapper around either a `AbstractValue` or a `AbstractMutVal` (the implementations of `PrimVal#` and `Object#`). Note that `AbstractMutVal` subclasses `AbstractValue`, as well as do `Int`, `Float`, `NoneType` and `Bool` which implement `Int#`, `Float#`, `None#` and `Bool#`, respectively, and also subclass from `AbstractValue`.

`AbstractValue` is an abstract class that defines all the operations supported (`+`, `*`, ...) by Pytropos, and what it is required for a function call, subscript access and attribute access. `AbstractValue`, in its turn, subclasses `AbstractDomain` a abstract class that defines the methods every Abstract Domain should have, namely `is_top()`, `join()`, `top()` and `widen_op()`. `PythonValue` and `Store`, unsurprisingly, also subclass `AbstractDomain`.

The figure below presents a class diagram showing the relationships between the different components in Pytropos.

[3]I use MyPy and flake8 in every project and I am thankful to the years of effort put into these amazing tools. Thank you guys!

[4]Both representations are explained in detail in the previous chapter



`<prim-callable>` and all other `Objects#` are implemented by a subclass of `AbstractMutVal`: `<prim-callable>` by `BuiltinFun`, `Module` objects by `BuiltinModule`, `lists` by `List`, and `tuple` by `Tuple`.

Chapter 5

Validation and Discussion

So, this chapter is about which kind of tests where performed and which were considered but didn't cut it. The Python Regression tests are very broad in scope, they use too many builtin capabilities which makes it hard for a prototype like Pytropos to make any of them work (and that is maybe why they are the regression tests for Python, after all they need to check as many characteristics as possible!).

5.1 Validation

I created two types of tests: unit regression tests and property-based tests. The unit tests let us check for any kind of anomalous behaviour while property-based tests ensure that the abstraction follows the Python implementation that underlies.

Comparing against the official regression tests is left for future work, as the number of builtin characteristics does not yet reach a satisfactory level to check.

5.1.1 Unit Tests

There are a total of 79 unit tests checking various parts of the Abstract Interpreter. The tests can be categorised into:

- binary operations
- type annotations
- numpy library
- if branching
- while looping
- lists and tuples
- join stores

In the table below the result of executing all tests can be seen. The coverage is not 100% as Pytropos is still in development. The tests check not only for what Pytropos is able to do now, but what it should be able given the description of it given in the document.

Category	Total lines	Tests	Time analysing	covered	non-covered	Total coverage
Binary ops	120	10	500ms	8	2	80%

5.1.2 Property-based Tests

Property-based tests that try to disprove a property. For example, a property of lists is that anytime you add a new element to a list of size `n` you will get a list with size `n+1`. I use property-based tests to check for correcness of Pytropos against Python on builtin operations for builtin values.

There are a total of 20 of such tests. Because each one of them is a property-based tests, they are tested against a 100 values that try to disprove the property. This kind of testing proved to be extremely helpful at the start of the development, as the "property-based tester" was able to find numerous counter examples to the properties. Python throws an exception on many cases of bad values, the "tester" was able to reliably (at least at the start) to come up with combinations of values that would go wrong when operated. Some of the most honorable mentions are:

- Zero-division error on expressions like `5 % 0` or `5 % False`
- Value error on expressions like `5 << -2`
- Excesive High-memory consumption on expressions like `5 << 10**10`
- Overflow error on expressions like `10**209 + 2.0`

5.2 Discussion

Pytropos is still in a very young age. It is not able to check many, many characteristics present in Python code (some of the biggest culprits are `for` statement, custom classes and objects, exception handling and the lack of builtin definitions), but when code is given to it that it can actually check.

Pytropos is able to check for the many common cases and mistakes that can be made when working with tensors. It is able to calculate the shape of tensors in a variety of circumstances (NumPy's `array` can evaluate almost anything as an array, therefore detecting the shape of any value passed to `array` is an accomplishment in itself. An example of why this is important can be seen in the library, where they define a stub for mypy to check of the library numpy, and what they do in the case of `array` is basically give up).

Chapter 6

Related Work

A big, widely used, and mature language like Python has no lack of Static Analysis tools. A big, widely used, and mature area like Machine Learning has no lack of people trying to build tools to make writing code for it easier. In this chapter, a brief overview of the many Static Analysis tools developed for Python code and the several approaches taken to solve the problem of tensor shapes.

6.1 Static Analysis in Python

The table below summarises the different tools and what they rely on:

```
Tool name/source |  Usage  | Analysis base  | Purpose                      |
----------------------------------------------------------------------------
@cannon.._2005   | Embeded | Type analysis  | Type checking                |
Pyflakes[ref]    | Linter  | NA             | Various checks               |
Pylint           | Linter  | NA             | Various checks               |
Pychecker        | Linter  | NA             | Various checks               |
MyPy             | Linter  | Type analysis  | Type checking                |
Enforce*+(defun) |         | Type analysis  | Type checking                |
Sagitta* (defun) |         | Type analysis  | Type checking                |
StyPy            | Linter  | Novel          | Type checking                |
Lyra           |        | AbstrInter    | Various analyses (including Value Analysis)   |
Pytropos       | Linter  | AbstrInter    | Value Analysis (specialised on tensor shapes) |
PyType           | Linter  |                | Type checking and inference |
ICBD (defun)     | Linter  |                | Type checking and inference |
Pyre             | Linter  | AbstrInter     | Type checking                |
Nagini           | Linter  | SMT (Viper)    | Verifier                     |
Typpete          | Linter  | SMT            | Type inference               |
fromherz...2018  | ?       | AbstrInter     | Value Analysis | Verifier    |
monat..2018      | ?       | AbstrInter     | Type Analysis                |
PyAnnotate*+     | Library | Type analysis  | Type inference               |

*: special cases, they are not an static analysis, but dynamic analysis!
?: tool has not been made public
+: uses gradual types

Pyflakes [@pyflakes2005]
Pylint [@thenaultpylint]
```

```
PyChecker [@norwitzpychecker]
Pytype https://github.com/google/pytype
ICBD https://github.com/kmod/icbd
Pyrecheck https://github.com/facebook/pyre-check
Sagitta https://github.com/peterhil/sagitta
Nagini https://github.com/marcoeilers/nagini (has paper)
Typpete https://github.com/caterinaurban/Typpete (two references, a thesis and a paper)
monat..2018 paper: Static Analysis by Abstract Interpretation Collecting Types of Python Programs
PyAnnotate https://github.com/dropbox/pyannotate
```

6.2 Tensor Shape Analysis

6.2.1 Solutions in other languages

We may think, If a type system is not expressive enough to capture the shape of tensors, then we should just start writing code in a language which does. We may write code in a language like Haskell or even C++, which have very strong and expressive type systems (it is possible in C++ to enforce the shape of the tensors with the use of templates and constraints (C++20)).

In fact, solutions in other languages exists. For example, Chen (2017) type checks the shape of tensors (operations) by restricting what can be constructed (via constraints in the types of objects and functions). Chen's solution uses the powerful type system of Scala (which runs over the JVM). Eaton (2006) does the same, although in an old version of Haskell. Eaton's encoding of tensor shapes is awkward because Haskell did not have, at the time, support for natural numbers at the type level. Haskell also lacked on syntactic sugar for type functions. Abe and Sumii (2015) implement type checking for matrices (aka, tensors) in OCaml. The library detects the shape of the matrices at compile time. Rakić, Stričević, and Rakić (2012) type check tensor shapes (they call them matrices) with templates in C++. Templates are only accessible at compile time, thus the Rakić et al. library type checks at compile.

One recent effort to type check tensor shapes in Haskell is the library `tensorflow-haskell-deptyped` written by Cruz-Camacho and Bowen (2018). The library is written as a wrapper around the library port of TensorFlow for Haskell. `tensorflow-haskell-deptyped` enforces at the type level how and which results a operation between compatible tensors is to be performed.

A different path to take is to extend an existing type checking system, like MyPy, and extend it with dependent types. Dependent types allow us to carry information from the term level to the type level, i.e. we can encode information of our data (available only at runtime) into the type system. Over this extended type system, we could implement some restrictions to the operations applied to different types (this is in fact the strategy taken in the library tensorflow-haskell-deptyped).

6.2.2 Theoretical Solutions

The problem of mismatched shapes is not new, in fact, it is so common that it has appeared several times with similar solutions (Arnold, Hölzl, Köksal, Bodík, & Sagiv, 2010; Griffioen, 2015; Rink, 2018; Slepak et al., 2014; Trojahner & Grelck, 2009). All solutions, though, are theoretical, they propose type systems which, if were to be implemented, could warranty type safety, i.e. no mismatching of tensor shapes could ever happen at runtime.

The following is a small discussion about the different solutions proposed to type check tensors found in the literature[1]:

- Trojahner and Grelck (2009): A paper on type checking of arrays. They define all type restrictions using dependent types (something that no other paper does). The special keyword of this work is "array programming".
- Arnold et al. (2010): A paper focused on type checking of sparse "matrix" operations. They define a functional language that can be translated into a lower level language, or machine code. They give a complete formalization of the algorithms and proofs of correctness in Isabelle. The special keyword of this work is "sparse matrix".
- Griffioen (2015): A paper focused in type checking and inference of arrays in array programming and vector spaces. They define a special type system in which tensors are first class citizens. The algorithm used for type checking is "Unification" which allows them to infer the type shape of arrays. The special keyword of this work is "array programming".
- Slepak et al. (2014): A paper that tries to formalise array-oriented programming languages and extend them with unit-aware operations. Array-oriented programming languages are languages which base all their computations in arrays (like matlab), but they usually aren't formalised. The types of arrays not only carry information about their shape but also of the unit they carry. The special keywords of this work are "array-oriented programming" and "unit-aware computation".
- Rink (2018): In this paper, Rink formalises a type system to type check the shape of tensors and defines a language to use with the type system. It is a custom type system which does not require of dependent types. The special keyword of this work is "tensor manipulation language".

[1]As a side note, it is interesting to notice how difficult is to communicate ideas in science. All the papers presented in this section hope to solve the same (or similar) problem, but they do not reference each other, which means that none of them knew what the others were doing. The principal reason for this, I think, it is because they all called the problem differently.

Chapter 7

Conclusions

In this work I have presented the application of a very powerfull and very well studied technique for Static Analysis (Abstract Interpretation), applied to a domain where it has not been applied much (just a couple of recent papers and applications are using it). In a brief note the work done includes:

- A formalisation of a subset of Python 3.6
- An Abstract Domain for Python Values
- An Abstract Domain for Python program states
- The semantics of an Abstract Interpreter for Python
- The implementation of the Abstract Interpreter
- A prove of concept domain where, tensor shapes, where the Abstract Interpreter could show its power
- A series of tests showcasing the scenarios where the AI is able to work fine and show results
- A way for developers to annotate code to improve the accuracy of the AI

The formalisation of the subset of Python showed one more attempt to formalise the Python semantics. We focused on how this formalisation could be used for a very specific library, the numpy library a corner stone of modern Python.

To implement the Abstract Interpreter, we required the formalisation of the Abstract Domain which carries the values stored in memory. The State Abstract Domain is wholy defined by the join operation. The join operation is the most complex piece in the whole work, it tries to combine the information of two different states into one that contains information from both. The values where they differ are then collapsed properly to a `Top` value. `Top` values can be think of as `?` types in Gradual Type systems, they are valid types/values but carry only the information of their existance not what could be they carrying.

Fortunately, the semantics of the Abstract Interpreter are not very complicated, they just borrow their capabilities from the State Abstract Domain and the semantics of Python. It is easy to see how the formalisation could be extended to work with other values by just augmenting the Abstract Domain.

"Extensive" testing showed that the implementation of the Abstract Interpreter, Pytropos, is able to actually check many common shape cases. The biggest problem of the implementation relies on the lack of capabilities it can offer (harshly limited to forgo user-defined functions and classes).

The Abstract Interpreter also lets itself to extend with type annotations. Type Annotations are meant to be given by the user, who always knows what is doing, and can tell the poor AI what the value of some variable is. This is easily done by replacing the old value by the new one with the annotation only if the annotation contains for information about the variable than the value contained on it (ie, hint $<$ val).

7.1 Future Work

Python is a huge and rich language. The amount of Python characteristics exceeds that of what a single human could try to implement for a work like this. In fact, implementing an Abstract Interprer can be regarded to be as hard, or harder, than a regular interpreter.

Much work left to do to try to match the power of Pytropos to make it usable for the regular developer. I propose the following roadmap to continue building the Abstract Interpreter:

- Extend Pytropos to include Exception handling (probably, a similar approach to that of could be applied here).
- Improve how spliting stores and **join**ing them latter is done. The join operation between stores is very, very costly. It requires walking through both graphs/stores at the same time. That is extremely expensive when only a variable has changed its value between two different stores. This associated cost could be reduced if not a whole store is created everytime a new store is need to be created. (One way to do this is by using immutable structures for all values in the implementation. The current implementation uses the Heap provided by CPython underneat but if one where to define ones heap from scratch this would not be a problem).
- Extend, again using the ideas put on the paper on ai , Pytropos to handle break statements.
- Extend the definition of Store from a simple Global scope + heap to something that handles different levels of scope. This would allow an easier implementation of user defined functions. The function scope of Python is middly complex (this is due to the `compile`-ation process to which the parsed code is put to. Python does compile, but it does it very, very fast) with four different variable scopes: global, local, nonlocal, and class.
- Once the interpreter handles user-defined functions properly, extending the interpreter to work with user-defined objects is not a big problem. The biggest difficulty is the inherit MRO behind the complex multi-classing system of Python.

Besides what is left to do to make Pytropos more powerful, there are some tasks related to the formality of the work. This work just presented a formalisation of a subset of Python, an Abstract Domain for Python, and "inferred" Abstract Semantics, but it nothing is ever proved. I built from theories to make something workable, but it may be good if we try to formalise a little bit more Python. We want some warranties after all! The things that I consider should be done next are:

- Give a complete and through formal definition for a subset of Python in a proof-assistant language such as Agda, Idris, Coq, Isabelle/HOL, or one of the many others.
- Define in the same proof-assistance language the Abstract Domain, the properties it must follow, and the inferred semantics it produces.
- Prove that the inferred semantics are in fact **inferred** from the Galois connection of the Abstract Domain.

Such a formalisation of Python would likely/hopefully help future endeavours on proofs and verification.

Notice that any formalisation of Python must take some stance on how closely it wants to follow the implementation of CPython. CPython has many, many little undocumented semantic subtleties that trying to write a full formal definition would probably defeat its purpose, namely, help other projects where a clean and clear formalisation is necessary.

Appendix A

Python Static Analysis based on Abstract Interpretation

This chapter is divided into two parts: first, I present a reduced set of Python syntax together with a (partial) formal definition of Python semantics, and later, I show the use of the formal semantic definition to define the abstract interpretation solution I implemented.

This chapter is meant to explain the theory behind the Abstract Interpreter implemented. In the next chapter an explanation of the implementation details is given[1].

A.1 Python (reduced) syntax and semantics

Python has no official formal semantics. The Python Software Foundation defines a Reference Manual for the language (Foundation, 2019), but they are explicit that the manual does not define a full specification for the language. Quote: "... if you were coming from Mars and tried to re-implement Python from this document alone, you might have to guess things and in fact you would probably end up implementing quite a different language."

There have been a couple of formal specifications defined for Python (Politz et al., 2013; Fromherz et al., 2018; Guth, 2013; Ranson et al., 2008). In this work we will define yet one more set of formal semantics for Python. I decided to define my own subset of Python because of my specific needs, some formalisations had more than what I needed and other lacked some aspect that I wanted.

I got to admit that my formalisation resembles the most that of Fromherz et al. (2018). In fact my formalisation is loosely based on Fromherz et al. (2018) althought the semantics of both are quiet different. As in Fromherz et al. (2018), we define each expression and statement step as a function from states of the program to states of the program, but opposed to them I do not include exception handling and breaking statements. My formalisation allows every single value in the language to be an object: functions, attributes, subscripts, and primitive values are all objects just as in regular Python. The following is valid in Python and in this formalisation:

```
a = □
b = a.append
b(3)
a.append(2.0)
print(a)  # prints: [3, 2.0]
```

In this sense, my formalisation is closer to that of Politz et al. (2013), where they also show how their formalisation is able to handle similarly complex examples.

[1]Opposed to how it looks like, the implementation came first. The formalisation has only been written specifically for this work.

A.1.1 (Reduced) Syntax

A simplified version of Python's Syntax. Modified from AST's syntax below.

```
mod = stmt*  -- A program. Starting point

expr = Int(i) for i \in \N | Float(j) for j \in floats
     | True | False | None

     | identifier         -- variable name

     | expr op expr       -- eg, a + 5
     | expr cmpop expr    -- eg, a < 5
     | expr(expr*)        -- Function calling

     | expr.identifier    -- Attribute access
     | expr[expr]         -- Not supported for Numpy Arrays :S
     | [expr*]            -- list
     | (expr*)            -- tuple

stmt = del expr           -- delete expression
      | expr = expr       -- assignment
      | expr op= expr     -- augmented assignment
      | expr: expr = expr -- type annotation
      | while expr: stmt+
      | if expr: stmt+
      | import alias+
      | from identifier import alias+
      | expr              -- An expression can be an statement

op = + | - | * | / | % | ** | << | >> | //
cmpop = < | ⩽ | > | ⩾

alias = identifier | identifier as identifier

identifier = string  -- with some restrictions
```

The syntax above is a subset of the Python 3.6 syntax. CPython does not directly interpret code written in the syntax above. The usual steps of lexing and parsing into a more explicit representation are necessary. We will define the semantics of the language over a reduced parser syntax and not the syntax defined above, as it eases the semantic definition of the language. [2]

```
mod = Module(stmt* body)

expr = Int(n)
     | Float(n) | True | False | None
     | Name(identifier, expr_context)
```

[2]Modified from the Python 3.6 syntax found at https://github.com/python/typed_ast/blob/89242344f18f94dc109823c0732325033264e22b/ast3/

```
     | BinOp(operator, expr, expr)
     | Compare(expr, cmpop, expr)
     | Call(expr, expr*)
     | Attribute(expr, identifier)
     -- No need for expr_context because no user made objects are allowed yet, thus
     -- modifying attributes is not necessary
     -- | Attribute(expr, identifier, expr_context)
     | Subscript(expr, expr, expr_context)  -- No arbitrary slice allowed yet
     | List(expr*)
     | Tuple(expr*) -- No expr_context for Tuple as `(a, b) = 1, 2` is not supported

stmt = Delete(expr+)
      | Assign(expr, expr)                -- a = 3
      | AugAssign(expr, operator, expr)   -- a += 3
      | AnnAssign(expr, expr, expr)       -- a: int = 3

      | While(expr, stmt+)
      | If(expr, stmt+, stmt*)

      | Import(alias+ names)
      | ImportFrom(identifier, alias+)

      | Expr(expr)

-- Indicates why are we looking up a variable, attribute or subscript
expr_context = Load | Store | Del

operator = Add | Sub | Mult | Div | Mod | Pow | LShift
             | RShift | FloorDiv

cmpop = Lt | LtE | Gt | GtE

-- import name with optional 'as' alias.
alias = (identifier, identifier?)
```

Code written in our subset of Python gets translated into this "AST" representation, over which we define the semantics of the language. We will no describe the process of translation as it is a very well studied subject (parsing), but will explore some important examples to show why the translation aids into the definition of the formal semantics of the language:

- `a + b` gets translated into `BinOp(Add, Name(a, Load), Name(b, Load))`. Notice the `Load` context, it indicates that we want to get the value of the variable not a reference to it (if we wanted to alter it).
- `a = 3` gets translated into `Assign(Name(a, Store), Int(3))`. Notice the `Store` context, it tells us that we will get a reference to where the value is stored and not its value.
- `del a` gets translated into `Delete(Name(a, Del))`. Notice the `Del` context, it tells us that we will get the object where it was called from as well as it possition in the heap.

- `a.b[3] + b.c` gets translated into

  ```
  BinOp(Add,
    Subscript(Attribute(Name(a, Load), b), Int(3), Load),
    Attribute(Name(b, Load), c)
  )
  ```

- `a.b[3] = 3` gets translated into

  ```
  Assign(
    Subscript(Attribute(Name(a, Load), b), Int(3), Store),
    Int(3)
  )
  ```

  Notice how we only get the `Store` context for the subscript and not for anything else, as we only want to know where the value of the subscript is stored and nothing else.

- `del a.b[3]` gets translated into

  ```
  Delete(
    Subscript(Attribute(Name(a, Load), b), Int(3), Del)
  )
  ```

  Notice how we only get the `Del` context for the subscript and not for anything eles, as we only want where and whom belongs to the subscript and nothing else.

We will see in a moment, but the semantics of a `del` require us to know where the identifier or attribute is located (in an object or the store). Assigning a variable require us to know where to put a variable. Accessing to innexistant attribute (`class A(): ... ; a = A(); a.length # error: attribute unknown`) it's not the same as to defining a new attribute (`class A(): ... ; a = A(); a.length = 3 # works!`), thus we need a way to distinguish between this three different stament-dependent expressions.

As you may have noticed already, my subset of Python does not have the ability to allow the definition of custom functions or classes. Despite the unability to define a custom function or class in the language, we want to be able to call a function and access to objects attributes. We have found that even though the amount of characteristics we support right now is small, we are able to capture some common errors caused when coding (e.g. `5 % 0` fails and we can capture it).

A.1.2 Python (reduced) small step semantics

There are two types of values (objects):

- Primitive values (`PrimVal`): integers (`ints`), Floating point numbers (`floats`), Boolean values (`True` and `False`), and the lonely `None` value.
- Mutable values: `Object`. An `Object` is a value that holds a type, an address to where is located, and a "dictionary" pointing to other values.

Lists, Tuples, builtin Functions, builtin Modules, builtin Classes, and builtin Methods derive from `Object`. Mutable Values may not allow to change the value of their attributes, as in the case with Tuples, the name "Mutable Values" refers to their ability to point at other values (either Primitive and Mutable) and possibly change them.

The **state of a Python program**, called the store of the program, is a tuple `(G, Heap)` where `G` is the "global scope" of variables, and `Heap` the heap (where all values are stored). Notice that we are ignoring the statements to execute in the state of the program.

Putting all together we get:

```
Global = Iden -> Addr + Undefined -- Global scope
Heap = Addr -> Val + Undefined   -- Heap

Key = Iden + (string x (Iden + PrimVal))
Type = List | Tuple | Module

PrimVal = Int | Float | True | False | None | Undefined
Object = Type x Addr x (Key -> Addr + Undefined)
Val = PrimVal | Object | <prim-callable>

<prim-callable> = <prim-append> | <prim-+-int> | <prim-+-float> | <prim-*-int> | ...
```

A `<prim-callable>` is a value that is builtin function in Python. The special value `Undefined` is used to signal unassigned values in `Heap`. If one tries to operate with an `Undefined` value the execution should halt, operating with `Undefined` values is forbidden as they never appear on Python (In Python, an `Undefined` value is an errorneous memory value, or an unassigned region of memory).

An `Iden` is a Python identifier. A Python identifier is a string that can only contain letters, numbers, and the character _. An identifier cannot start with a number[3].

Notice how the index (`Key`) for the function that relates an `Object` to its attributes can be one of two things. `Key` is either an `Iden` or a tuple `string x PrimVal`. The idea behind indexing an object with two separate kinds of keys is to be able to diffirenciate between a value that is inherit to the `Object` and other that the object simply points. Consider a list, an inherit, unmodifiable, value of a list is its size. The size of a list can only be modified if an element is added or removed from it. Now, consider the value at the index 2 of the list `[2, 54, [True], 6, 0.0]`, the value is another `Object`. Any object stored in a list is not an intrisic property of the list.

A `Key` can be a tuple `string x PrimVal`. There are only two types of tuples in the current formalisation, either `('index', val)` or `('attr', val)`. A key of the form `('attr', val)` indicates us that `val` (hopefully an `Iden`) is an attribute of the object. `('index', val)` is used for lists.

For example, the list `[None, 4, ()]` can be expressed as:

```
(List,
 0,
 { 'size': 1,
   ('index', 0): 2,
   ('index', 1): 3,
   ('index', 2): 4
```

[3]I am simplifying here for the sake of brevity. In fact Python 3 does allow a wide array of Unicode characters to construct an identifier. https://docs.python.org/3/reference/lexical_analysis.html#identifiers

```
 }
),
```

The first element of the triple is `List`, indicating us that the `Object` is a list. The second element is the address on the heap, a natural number. The third element is a function from `Key` values to `Vals`. Strictly speaking an `Object` cannot be defined isolated, it requires to be defined as part of a `Heap`:

```
H = {
  0: (List, 0, { 'size': 1, ('index', 0): 2, ('index', 1): 3, ('index', 2): 4}),
  1: 3,
  2: None,
  3: 4,
  4: (Tuple, 4, {'size': 5}),
  5: 0
}
```

Notation: A function is defined as a Python Dictionary. It is slightly easier to type and understand `{x: m, y: n}` than $x \to m; y \to n$.

Semantics of Expressions

In the same manner as Fromherz et al. (2018), we define the semantics of an `E[`*expr*`]` as a function that takes a state and returns a state plus a value.

An expression takes as inputs: * A Global Scope, and * A Heap

and the result of executing an expression is:

- A new Global Scope,
- A new Heap, and
- One of three things: A `Val`, an `Addr` or `(Object+None)xIden` if the context was `Load`, `Store` or `Del`, respectively.

```
E[expr] :: Global x Heap
        -> Global
         x Heap
         x (Val + Addr + (Object x (string x Val)) + Iden)

E[Name(id, ctx)](G, H) :=
  match ctx in
    case Load  -> if G(id) = Undefined
                  then <Execution Halt>
                  else (G, H, G[id])
   case Store -> (G, H, id)  -- Something will be stored in id S[Assign( ... )] or variation will take care o
    case Del   -> (G, H, id)  -- The id will be deleted, S[Delete( ... )] will take care of it

E[BinOp(op, a, b)](G, H) :=
  let (G1, H1, v1) := E[a](G, H)
      (G2, H2, v2) := E[b](G1, H1)
   in if kind(v1) /= Val  or  kind(v1) /= Val
      -- Bad parsing! `E[a](G, H)` is supposed to return Val
      then <Execution Halt>
      else
        let prim_op := get_prim_op(op, type(v1), type(v2))
         in prim_op(v1, v2, G2, H2)

E[Attribute(e, attr)](G, H) :=
 let  (G1, H1, ad) := E[e](G, H)
      -- `e` must compute to a Val, we don't need Addr's or an Iden to delete
      v  := if not is_value(ad) then <Execution Halt> else ad
 in match v in
      case v: PrimVal ->
        -- primitive, similar how get_prim_op is coded
        get_prim_attr(type(v), atr)(G1, H1, v)

      case (t, addr, o): Object ->
        -- ALL values in the current definition are builtin
        if builtin(t)
        then get_prim_attr(t, attr)(G1, H1, v)

        -- Accessing (non builtin) value's attributes never happens.
        -- This code is left to show how we plan to expand the current
        -- system to support attribute access for custom objects
        else let  addr := o('attr', attr)
              in  if addr = Undefined
                  then <Execution Halt>
                  else (G1, H1, H1(addr))

      case <prim-callable> ->
        (G1, H1, v)


E[Subscript(e, i, ctx)](G, H) :=
 let  (G1, H1, ad) := E[e](G, H)
      v  := if not is_value(ad) then <Execution Halt> else ad
      (G2, H2, ind) := E[i](G1, H1)
  in
     match ctx in
       -- A Subscript with Load always returns a Val
       case Load ->
         match kind(v) in
           case (_, _, o): Object ->
             let  addr := o('index', ind)
              in  if addr = Undefined
                  then <Execution Halt>
                  else (G2, H2, H2(addr))

          otherwise -> <Execution Halt> -- No PrimVal or <prim-callable> is subscriptable
       -- A Subscript with Store always returns a (Object x (string x PrimVal))
       case Store ->
         match v in
           -- There is one check left to do, ind should be a prim val
           case Object  -> (G2, H2, (v, ('index', ind)))
           otherwise -> <Execution Halt>

       case Del ->
         if kind(v) = Object
         then (G1, H1, (v, ('index', ind)))
         else <Execution Halt>


E[List(lst)](G, H) :=
  let freeaddr := get_free_addr(H)
      empty_lst_fun('size') := length(lst)
      empty_lst_fun('index', n) :=
        if n < length(lst)
        then lst[n]  -- abusing notation, taking the `n` value from the list
        else Undefined
      lst := (List, freeaddr, empty_lst_fun) -- An object is a tuple
   in (G, H[freeaddr->lst], lst)

E[Call(caller, vals)](G, H) :=
  match E[caller](G, H) in
    -- Abusing notation by magically unfolding `vals`
    case (G1, H1, call: Val) -> call(*vals, G, H)

    otherwise -> <Execution Halt>  -- the caller must be a Val
```

`<Execution Halt>` is used in two ways in here. Either it means that we found an operation that throws an exception (which this formalisation does not handle), or it means that the AST is malformed and nothing can be further calculated (an example of this is using the wrong context, e.g. `Load` when the value required a `Store` context).

Note: Regarding the semi-casual notation used in here, `type(Something)` is meant to be a shorthand to expanding on the definition of `Something`. The porpuse is to make the code a little bit more intelligible.

Notice that we make use of `get_prim_op` to find the appropiate primitive function to operate two different values. Later, when we extend Python with NumPy arrays will extend `get_prim_op` to work with them.

```
TypeVals = Type U {Int, Float, Bool, NoneType}
get_prim_op :: Op x TypeVals x TypeVals -> <prim-callable>
-- Type can be Int, Float, List, ...

get_prim_op(Add, t1, t2) :=
  match (type(v1), type(v2)) in
```

```
    case (Int, Int) -> <prim-+-int>
    case (Float, Float) -> <prim-+-float>
    case (Int, Bool) -> \(i, j, G, H)-> <prim-+-int>(i, Int(j), G, H)
    case (Bool, Int) -> \(i, j, G, H)-> <prim-+-int>(Int(i), j, G, H)
    case (Float, a) -> if a = Bool or a = Int
                       then \(i, j, G, H)-> <prim-+-float>(i, Float(j), G, H)
                       else <Execution Halt>
    case (a, Float) -> if a = Bool or a = Int
                       then \(i, j, G, H)-> <prim-+-float>(Float(i), j, G, H)
                       else <Execution Halt>
    -- This function is to be extended once we add NdArrays to the mix
    case otherwise -> <Execution Halt>
```

As an example, `<prim-+-int>` is defined as the function:
`<prim-+-int>(i, j, G, H) := (G, H, i+j)`

Statements Semantics

The semantic of statements is a function between the state of the program, just like it was done with expressions. Unlike with expresions, the semantics of statements do not return any kind of value, they just modify the state of the program.

```
S[Assign(var, val)](G, H) :=
  let (G1, H1, ass) := E[var](G, H)
      (G2, H2, rightval) := E[val](G1, H1)
      rval := if is_value(rightval) then val else <Execution Halt>
   in
      match ass in
        case Iden -> (G2[ass->rval], H2)

        case ((t, addr, o): Object, ('index', val: Val)) ->
          let setindex := get_prim_set_index(t)
          in setindex(G2, H2, o, val, addr, rval)

        -- This case doesn't come up, it is only required when when user objects
        -- are allowed
        -- case ((t, addr, o): Object, ('attr', val: Val)) ->
        --    let

        otherwise -> <Execution Halt>

-- Behaviour in Python 4
S[AnnAssign(var, hint, val)](G, H) := S[Assign(var, val)](G, H)

-- Behaviour in Python 3
S[AnnAssign(var, hint, val)](G, H) :=
  let (G1, H1, evaluatedhint) := E[hint](G, H)  -- In Python 3 the hint is computed
   in S[Assign(var, val)](G1, H1)
```

```
get_prim_set_index : Type
                   -> type(G) x type(H) x (Key -> Undefined + Addr) x Val x Addr x Val
                   -> type(G) x type(H)
get_prim_set_index(List)(G, H, o, ind, addr, rval) :=
  if kind(ind) /= Int
  then <Execution Halt>
  else if 0 ⩽ ind and ind < o('size') -- negative cases can be added later
  then
     let newlst := (List, addr, o[ind->rval])
      in (G, H[addr->newo])
  else <Execution Halt>
get_prim_set_index(Tuple)(G, H, o, ind, addr, rval) := <Execution Halt>
get_prim_set_index(_)(G, H, o, ind, addr, rval) := <Execution Halt>

S[Delete(e)](G, H) :=
  let (G1, H1, a) := E[e](G, H)
   in
      match a in
         case Val -> <Execution Halt>
      case Addr -> <Execution Halt> -- e should have returned a way to find the place to remove the value
         case Iden -> (G[e -> Undefined], H)
         case ((type, addr, o): Object, key: (string x Val)) ->
           let del := get_prim_delete(type, key)
            in del(G1, H1, o, addr)

get_prim_delete(List, key) :=
  match key in
    case ('index', val: Val) ->
      <prim-del-index-list>(val)
    otherwise -> <Execution Halt>
get_prim_delete(Tuple, key) := <Execution Halt>
-- other get_prim_delete could be added, for example if attributes could be deleted (only
-- with user defined objects)

<prim-del-index-list> : Val
                      -> Global x Heap x (Key -> Undefined + Addr) x Addr
                      -> Global x Heap
<prim-del-index-list>(ind)(G, H, lst, addr) :=
  if type(ind) /= Int
  then <Execution Halt>
  else
     if ind < lst('size') and ind ⩾ 0 -- Other cases to handle are when ind < 0, in Python that is valid!
       then let newlst1 := shift-left-ind-in-list(lst, ind, lst('size'))
                newlst2 := newlst1[('index', size-1)->Undefined]
             in (G, H[addr->newlst2])
       else <Execution Halt>
```

```
shift-left-ind-in-list(lst, ind, size) :=
  if ind < size - 1
  then shift-left-ind-in-list(lst[('index', ind)->lst('index', ind+1)], ind+1, size)
  else lst

-- Import will be defined later once we introduce the NumPy library
S[Import(name)](G, H) := <Execution Halt>
```

Notice that type annotations behave differently in Python 4 to Python 3.6+. Type annotations in Python 3.6+ are just regular expressions in the language, are evaluated and can modify the state of the program. For Python 4, it is planned that Type annotations will not modify the state of the program[4] but must obey the syntax of Python . In Python 3.7 a `__future__` import was added to modify the behaviour/semantics of Type Annotations for Python 3.7+. One can add `from __future__ import annotations` at the start of the file to forgo the evaluation of type annotations. We assume in this work that the type annotations do not alter the state, i.e. we assume that the user implicitely or explicitely is using the `annotations`' future statement.

In this formalisation, the delete statement is only able to delete variables in the global scope, but cannot delete attributes of an object. This limitation comes from the fact that, the formalisation lacks the capacity to define user-defined classes. Future work will focus on extending the state model to include function variable scope, and the ability to define functions and classes.

A.1.3 NdArrays

The purpose of the formalisation is to be able to construct from it an Abstract Interpreter. To test the Abstract Interpreter abilities to find bugs it should be able to handle NumPy array (tensors). In this subchapter, I extend the formalisation with NumPy array and discuss some of their semantics.

The following are all the things to take into account when extending our formalisation to handle a new type of "builtin" `Object` type:

1. Extend the types of `Objects` to handle NumPy arrays.

   ```
   Type = List | Tuple | Module | NdArray
   ```

   As an example, consider the numpy array `np.zeros((4, 3))`, it can be expressed as:

   ```
   Object(NdArray,
    0×anumber,
    -- The shape of a NdArray is a tuple of integers
    { 'shape' -> Object(Tuple,
                  0xothernum,
                  { 'size' -> 2,
                    ('index', 0) -> 4,
                    ('index', 1) -> 3,
                  }
   ```

[4] Well, this is not strictly true. In both, Python 3.6+ and Python 4, type annotations are stored in the special variable `__annotations__`.

```
              )
   ('index', 0) -> 0,
   ... -- all other indices, each one identified by an integer
 }
)
```

Note: Remember that an `Object` is a triple of the form `Type x Addr x (Key → Addr + Undefined)`. Also remember that example above is faulty as the codomain of the function defined above is `Val` not `Addr` as it should be.

2. Extend primitive functions with NumPy's primitive functions. The NumPy's primitive functions implemented in the Abstract Interpreter are: `array`, `zeros`, `dot`, `ones`, `abs` (and all other functions that don't alter the shape of the tensor they take), `arange`, `size`, `ndim`, `astype`, and `T`.

```
<prim-callable> = ... (old operations) |
                  <prim-np-zeros> | <prim-np-dot> | <prim-np-abs> | ...
```

Note: The values stored inside a NumPy array are consider irrelevant in this work. The Value Analysis built in this work considers only the shape of tensors, as tensors can be huge and their contents do not often influence their shape. Therefore, it would be wasteful to give a detailed formalisation of the NumPy library primitives.

Nonetheless, defining formaly each one of the NumPy functions above is fairly straightforward. Although, the hardest part of a formal defitinion of Numpy arrays is detailing how `array` works. To define the function `<np-array>` one must consider the many input cases it can handle, and it can handle almost any Python object[5].

Once the `<np-array>` function is implemented all other functions are much simpler to define. As an example, the implementation of the function `size` is:

```
<prim-np-size>(val)(G, H) :=
   -- We know that `<prim-array>` always returns an NdArray
   let (G, H, (NdArray, addr, arr)) := <prim-array>(val)(G, H)
   -- We know that a NdArray has a special value called `shape`
       (Tuple, addrtup, tup) := arr('shape')
   in  tup('size')
```

3. Extend the cases that `get_prim_op` handles to cover NumPy arrays. All operations defined in NumPy handle broadcasting. For example:

```
get_prim_op(Add, t1, t2) :=
  match (type(v1), type(v2)) in
    ... -- old cases
    case (NdArray, t1) ->
      \(i, j, G, H)->
```

[5]The NumPy function `array` takes almost anything as an input. `arrays` tries to interpret its input as an array in any way it can. There is no formal definition of how the values are interpreted althought its semantics can be extracted by looking at its C implementation: https://stackoverflow.com/a/40380014

```
          let (G2, H2, ndarr) := <prim-array>(j)(G, H)
          in  <prim-+-ndarray>(i, ndarr, G2, H2)
      case (t1, NdArray) ->
        \(i, j, G, H)->
          let (G2, H2, ndarr) := <prim-array>(i)(G, H)
          in  <prim-+-ndarray>(ndarr, j, G2, H2)
      case otherwise -> <Execution Halt>
```

4. The NumPy module holding all operations is defined:

```
<numpy-mod> := Object(Module,
 -1,  -- This value will be changed once it is imported
 { ('attr', 'array') -> <prim-array>,
   ('attr', 'dot')   -> <prim-dot>,
   ('attr', 'zeros') -> <prim-zeros>,
   ('attr', 'ones')  -> <prim-ones>,
   ...
 }
)
```

5. And finally, `S[Import(name)]` is extended (now it handles a single library):

```
S[Import(name)](G, H) :=
  match name in
    ("numpy",) ->
      let (Module, arr, mod) := <numpy-mod>
          freeaddr := get_free_addr(H)
      in  (G['numpy'->freeaddr], H[freeaddr->(Module, freeaddr, mod)])
    ("numpy", alias) ->
      let (Module, arr, mod) := <numpy-mod>
          freeaddr := get_free_addr(H)
      in  (G[alias->freeaddr], H[freeaddr->(Module, freeaddr, mod)])

    otherwise -> <Execution Halt>
```

A.2 Abstract Interpreter

We have the base to build an Abstract Interpreter, we have the semantics of Python (what is a variable, what is the state of the program, and how to modify the program (its formal semantics)).

The steps to build an Abstract Interpreter are:

- Define a Variable Abstract Domain,
- Define a State Abstract Domain, and
- Define the abstract semantics for the language.

A.2.1 Variable Abstract Domain

Remember, the possible values that a variable may have in Python are:

```
PrimVal = Int | Float | True | False | None | Undefined
Object = Type x Addr x (Key -> Addr + Undefined)
Val = PrimVal | Object | <prim-callable>

Type = List | Tuple | Module | NdArray
```

The definition of `Val` is a recursive but not the definition of `PrimVal`. We will start defining an Abstract Domain for `PrimVals` and later we will expand on it to define a "recursive" definition for the Abstract Domain of `Vals`.

`PrimVal` Abstract Domain

`PrimVal` is composed of five different types: `int`, `float`, `bool`, `NoneType`, and `Undefined`. We can define individual non-relational Abstract Domains[6] (AD) for each of the types.

The simplest of all AD is that of `NoneType`. `None` is the only inhabitant of `NoneType`. There is only one order for a set of one element, the trivial order: None $\leq$ None. The Galois connection for a trivial order is also very simple: $\alpha(None) = None^{\#}$ and $\gamma(None) = None^{\#}$.

A little bit more interesting is the AD for `bool`. `bool` is inhabited only by `True` and `False`. We define the following order:

```
    Top_{Bool#}
     /      \
  True#    False#
     \      /
    Bot_{Bool#}
```

The Galois connection for this lattice is also quite simple:

$$
\begin{aligned}
\alpha\colon \mathcal{P}(\text{Bool}) &\to \text{Bool}^{\#} \\
\emptyset &\mapsto \bot^{\text{Bool}} \\
\{True\} &\mapsto True^{\#} \\
\{False\} &\mapsto False^{\#} \\
\{True, False\} &\mapsto \top^{\text{Bool}}
\end{aligned}
$$

Where γ is just defined as α^{-1} given α's bijectivity.

Notice how our previous two Abstract Domains do not require us to define widening or narrowing operators because none of them has a

[6]A relational Abstract Domain is an Abstract Domain where the value of variables is not assumed to be independent of each other. For more information on relational Abstract Domains look at . All Abstract Domains used in this work are non-relational Abstract Domains as they are simpler to understand and implement. Future work will include extending the arange of Abstract Domains to use to some relational Abstract Domains.

For `int` and `float` there a plenty different different options. For some of them take a look at . We are going to use here, probably, the simplest AD for number systems there is Constant Propagation .

Constant Propagation is very simple, in fact both `None#` and `Bool#` are Constant Propagation Abstract Domains. We define `int`'s AD as:

```
                  Top^Int
  _________________|______________
 /     /     /          \     \    \
 0     1    -20 ...
 \_____\_____\__________/_____/____/
                   |
                  Bot^Int
```

Notation: To keep things light, $n^{\#}$ is represented as n.

In the same manner as with Bool$^{\#}$, we define the Galois connection as:

$$\begin{aligned} \alpha\colon \mathcal{P}(\text{Int}) &\to \text{Int}^{\#} \\ \emptyset &\mapsto \bot^{\text{Int}} \\ \{n\} &\mapsto n \\ otherwise &\mapsto \top^{\text{Int}} \end{aligned}$$

and

$$\begin{aligned} \gamma\colon \text{Int}^{\#} &\to \mathcal{P}(\text{Int}) \\ \bot^{\text{Int}} &\mapsto \emptyset \\ n &\mapsto \{n\} \\ \top^{\text{Int}} &\mapsto \text{Int} \end{aligned}$$

and

```
Int# = Int U {Top_int, Bot_int}
```

We define Float$^{\#}$ just in the same way.

Now that we have an Abstract Domain for each of `PrimVals` we can construct an AD for `PrimVal`. The idea is simple, as shown in the image below we just define an Abstract Domain that groups them all together and puts a value on top and one below all of them:

```
                  Top_primvals
      _________________|______________
     /            /                   \
   Top_int        Top_float
 ____|____     ____|____
/  /   \  \   /  /   \  \   Bool  Undefined NoneType
```

```
0  1   ...   .4 nan  ...
\__\___/__/   \__\___/__/
     |            |
   Bot_int       Bot_float
     \_____________\__________________/
                       |
                  Bot_primvals
```

The formal definition is quite simple, let's consider only the Galois connection as everything is quite straightforward:

$$\begin{aligned}\alpha^{\text{PrimVal}} : \mathcal{P}(\text{PrimVal}) &\to \text{PrimVal}^{\#}\\ \emptyset &\mapsto \bot^{\text{PrimVal}}\\ \{n\} &\mapsto \text{check n and create the according value given the n type}\\ otherwise &\mapsto \text{check if all elements belong to the same type (then same alpha) otherwise Top primval}\end{aligned}$$

This is not the only way to define an Abstract Domain out of other Abstract Domains. In fact there are many ways, on of which is to define an Abstract Domain where each is extended with `Undef` and they are all packed into a tuple .

`Vals` Abstract Domain

`Val` definition

Remember `Val`'s definition:

```
Val = PrimVal | Object | <prim-callable>
Object = Type x Addr x (Key -> Addr + Undefined)

Type = List | Tuple | Module | NdArray
Key = Iden + (string x (Iden + PrimVal))

Heap = Addr -> Val    -- Heap
```

A couple of important details about `Val`'s definition:

- A `Val` can be a `PrimVal`, an `Object` or a `<prim-callable>`.
- A `Val` is not isolated, it makes part of a bigger set of variables, all of them must be defined in `Heap`. Any `Val` we define must be stored in `H` $\in$ `Heap`.
- We say that a $(a, H) \in$ Addr $\times$ Heap is a **valid** value if every value defined in H: $vars = \{v \in Img(H) : v \neq \text{Undefined}\}$ is reachable from $H(a)$, and no `Addr` inside any defined `Object` points to `Undefined`.

An example of a possible value is `(0, H)` where `H` is defined as:

```
H = {
  0: (List, 0, { 'size': 1, ('index', 0): 2, ('index', 1): 3, ('index', 2): 4}),
  1: 3,
  2: None,
  3: 4,
  4: (Tuple, 4, {'size': 5}),
  5: 0
}
```

Notation: As in the previous subchapter we are using the Python dictionary notation to define functions. `{0: 4, 1: 5}` means $0 \mapsto 4; 1 \mapsto 5$

We shall remember this example from the previous subchapter, it represents the list `[None, 4, ()]`.

Notice that the values $(1, H)$, $(2, H)$, ..., and $(5, H)$ are not considered valid as it is impossible from them to reach all other values in the Heap. For this section, all values will be valid values. If we find a non-valid value (n, H), we can define a new value (n, H') where H' has all non-reachable values removed.

$Val^{\#}$ definition

An Abstract Value is a tuple $(a, H^{\#}) \in (Addr) \times (Val)^{\#}$ where:

```
Heap# = Addr -> Val# + Undefined    -- Abstract Heap

Val# = PrimVal# | Object# | <prim-callable># | Top_Val | Bot_Val
Object# = Type x Addr x ((Key -> Addr + Undefined) + ImTop + ImBot)
```

Notice that a `Val#` requires a Heap to work! Just as `Val` required it. Some examples of Abstract Values are $(a, H^{\#}) \in Val^{\#} \times Heap^{\#}$ are:

```
(0, {0: 3})
(0, {0: Top_Int})
(0, {0: Bot_Bool})
(0, {0: Top_Val})

(0, {
  0: (List, 0, { 'size': 1, ('index', 0): 2, ('index', 1): 3, ('index', 2): 4}),
  1: Top_Int,
  2: (Tuple, 2, ImTop),
  3: 21,
  4: (Tuple, 4, {'size': 5}),
  5: Bot_Int
})
```

Notice that we can represent any valid value $(a, H^{\#})$ as a graph with a as root:

```
({1}, {1: 3}, {(1,1): Undefined}, 1)
  (3)

-- if a pair (a, b) \in V x V is not shown, then it is Undefined
({1}, {1: Top_Int}, {}, 1)
  (Top_Int)
```

```
({1}, {1: Top_Bool}, {}, 1)
  (Bot_Bool)

({1}, {1: Top_Val}, {}, 1)
  (Top_Val)

(0, {
  0: (List, 0, { 'size': 1, ('index', 0): 2, ('index', 1): 3, ('index', 2): 4}),
  1: Top_Int,
  2: (Tuple, 2, ImTop),
  3: 21,
  4: (Tuple, 4, {'size': 5}),
  5: Bot_Int
})

({0, 1, 2, 3, 4, 5, 6},
 {0: List,
  1: Top_Int,
  2: Top_Tuple,
  3: 21,
  4: Tuple,
  5: Bot_Int,
  },
  {(0,1): 'size',
   (0,2): ('index', 0),
   (0,3): ('index', 1),
   (0,4): ('index', 2),
   (4,5): 'size'
   },
   1)

  (List)
   |-- 'size'       -> (Top_Int)
   |-- ('index', 0) -> (Top_Tuple)
   |-- ('index', 1) -> (21)
   |-- ('index', 2) -> (Tuple)
                         |-- 'size' -> (Bot_Int)
```

We have defined the first ingredient of the `Val` Abstract Domain. The ingredients left are:

- Abstraction α and concretisation γ functions,
- an order relation,
- join ($\sqcup^{\text{Val}}$) and merge ($\sqcap^{\text{Val}}$) operations, and
- a Galois connection.

$\cup^{\mathbf{Val}^{\#}}$ definition

We will start by the defining the *join* operation and the rationale behind its innerworkings. All other operations and functions are constructed in a very similar way as *join* is defined.

```
Uval: (Addr x Heap#) x (Addr x Heap#) -> (Addr x Heap#)
(n, H1#) Uval (m, H2#) :=
  let on := H1#(n)
      om := H2#(m)
   in if on is Object# and om is Object#
      then let (n', joined, Hnew#) := joinVal((n, H1#), (m, H2#), join_empty, H_empty)
            in (n', removeallInConstruction(Hnew#, joined, H1#, H2#))
      else if on is PrimVal# and om is PrimVal#
      then (0, H_empty#[0->on UPrimVal# om])
      else if on = Bot_Val
      then (0, H_empty#[0->om])
      else if om = Bot_Val
      then (0, H_empty#[0->on])
     else if on = om  -- checking all other cases TopVal = TopVal, <prim-_> = <prim-_>, ...
      then (0, H_empty#[0->on])
      -- the last case is when the two values have different types altogether
      else (0, H_empty#[0->Top_Val])

join_empty: Addr x Addr -> (Addr + Undefined)
join_emtpy(a,b) := Undefined

H_empty: Heap#
H_empty(a) := Undefined
```

join revises the kinds[7] of both values and defines a value that unifies them, they follow the following sensible rules:

- `Bot_Val` must be the lowest value in the order, therefore any value joining with it should be the same value (`Bot_Val U n = n`).
- `Top_Val` is the biggest value in the order, therefore any value joining with it should give back `Top_Val` (`Top_Val U n = Top_Val`).
- Values of the same kind, `PrimVal#`, `Object#` and `<prim-callable>`, should not be comparable, e.g. any value from `PrimVal#` joined with any of `Object#` should give `Top_Val`.
- Joining `PrimVal#`s should use `UPrimVal#`.
- Joining `Object#`s should take into account the recursive nature of the defitinion of `Val#`s and `Object#`s.

Notice tha it is `joinVal` where the whole magic of this Abstract Domain lies. `joinVal` is meant walk through both graphs simultaneously, find the similarities, implode the differences between the graphs and preserve the equaly looking parts.

The definition of `joinVal` requires the help of the function `joinObjectFn` which joins the values to which an `Object#` points (the function (`Key → Addr + Undefined`)).

[7] check if this is the right word to use

```
HeapCon# = Addr -> Val# + Undefined + InConstruction

-- we assume that (n,H1#) and (n,H1#) are Object#s
joinVal: (Addr x Heap#) x (Addr x Heap#) x (Addr x Addr -> Addr + Undefined) x HeapCon#
       -> Addr x (Addr x Addr -> Addr + Undefined) x HeapCon#
joinVal((n,H1#), (m, H2#), joined, H_new#) :=
  let joined_left := {l \in Addr | E r \in Addr : joined(l,r) ≠ Undefined}
      joined_right := {r \in Addr | E l \in Addr : joined(l,r) ≠ Undefined}

   in if n in joined_left or m in joined left -- n or m has alredy been visited
    then  if joined(n, m) ≠ Undefined -- an address for the new Object# has already been defined
            then (joined(n,m), joined, H_new#)
           -- either n or m had already been joined to other object, every variable they
            -- can reach should be Top_Val because the paths to reach them are different
            -- in the two states
        else let (joined', H_new'#) := makeallreachabletop((n, H1#), (m, H2#), joined, H_new#)
                     ad := freeaddr(H_new'#)
                  in (ad, joined[(n,m)->add], H_new'#[ad->Top_Val])
      -- we know that H1#(n) and H2#(m) are both 'Object#'s because the call from UVal#
      -- checked so
      else let -- n and m haven't already been visited
             (tn, adn, fn) := H1#(n)
             (tm, adm, fm) := H2#(m)
        in if tn ≠ tm -- the two Object#s are not of the same type
       then let (joined', H_new'#) := makeallreachabletop((n, H1#), (m, H2#), joined, H_new#)
                    ad := freeaddr(H_new'#)
                 in (ad, joined[(n,m)->add], H_new'#[ad->Top_Val])
           -- both objects have the same type
           else if fn = ImTop and fm = ImTop
                then let ad := freeaddr(H_new'#)
                      in (ad, joined[(n,m)->ad], H_new#[ad->(tn,ad,ImTop)])
                else if fn = ImTop
           then let (joined', H_new'#) := makeallreachabletop_right((m, H2#), joined, H_new#)
                          ad := freeaddr(H_new'#)
                      in (ad, joined'[(n,m)->ad], H_new'#[ad->Top_Val])
                else if fm = ImTop
           then let (joined', H_new'#) := makeallreachabletop_left((n, H1#), joined, H_new#)
                          ad := freeaddr(H_new'#)
                      in (ad, joined'[(n,m)->ad], H_new'#[ad->Top_Val])
                -- if any of the two values is Bot then we leave the new value as
                -- InConstruction until the end of the execution
                else if fn = ImBot or fm = ImBot
                then let adnew := freeaddr(H_new#)
                      in (adnew, joined[(m,n)->adnew], H_new#[adnew->InConstruction])
                -- both, fn and fn, are (Key -> Addr + Undefined)
               -- notice that `InConstruction` is assigned to the address `ad`, but once
                -- we return from the recursive call we can now replace the value for a
                -- proper object definition
                else let ad := freeaddr(H_new#)
                         (fnew, joined', H_new'#) := joinObjectFn((fn, H1#),
                                                                  (fm, H2#),
                                                                  joined[(n,m)->ad],
                                                             H_new#[ad->InConstruction])
                      in if H_new'#(ad) = InConstruction
                         then (ad, joined', H_new'#[ad->(nt, ad, fnew)])
                         else (ad, joined', H_new'#)

joinObjectFn: ((Key -> Addr + Undefined) x Heap#) x ((Key -> Addr + Undefined) x Heap#)
            x (Addr x Addr -> Addr + Undefined) x HeapCon#
            -> (Key -> Addr + Undefined) x (Addr x Addr -> Addr + Undefined) x HeapCon#
joinObjectFn((fn, H1#), (fm, H2#), joined, H_new#) :=
   let fn_empty: (Key -> Addr + Undefined)
       fn_empty = Undefined

       PreIm: (Key -> Addr + Undefined) -> P(Key)
       PreIm(fun) := {ad \in Addr | fun(ad) ≠ Undefined}

       helper: ((Key -> Addr + Undefined) x (Addr x Addr -> Addr + Undefined) x HeapCon#)
                x Key
              -> ((Key -> Addr + Undefined) x (Addr x Addr -> Addr + Undefined) x HeapCon#)
       helper((fnew, joined, H_new#), key) :=
         if fm(key) = Undefined
      then let (joined', H_new'#) := makeallreachabletop_left((fn(key), H1#), joined, H_new#)
                 ad := freeadd(H_new'#)
              in (fnew[key->ad], joined', H_new'#[ad->Top_Val])
         else if fn(key) = Undefined
      then let (joined', H_new'#) := makeallreachabletop_right((fm(key), H2#), joined, H_new#)
                 ad := freeadd(H_new'#)
              in (fnew[key->ad], joined', H_new'#[ad->Top_Val])
         -- key is defined in both, fn and fm
         else let
            adn := fn(key)
            adm := fm(key)
            val1 := H1#(adn)
            val2 := H2#(adm)
            ad := freeadd(H_new'#)

         -- same code that was in
         in if val1 is Object# and val2 is Object#
           then let (n', joined', H_new'#) = joinVal((n, H1#), (m, H2#), joined, H_new#)
                  in (fnew[key->n'], joined', H_new'#)
            else if val1 is PrimVal# and val2 is PrimVal#
           then (fnew[key->ad], joined[(adn,adm)->ad], H_new#[ad->val1 UPrimVal# val2])
            else if on = Bot_Val
            then (fnew[key->ad], joined[(adn,adm)->ad], H_new#[ad->InConstruction])
            else if val2 = Bot_Val
            then (fnew[key->ad], joined[(adn,adm)->ad], H_new#[ad->InConstruction])
              else if on = val2  -- checking all other cases TopVal = TopVal, <prim-
_> = <prim-_>, ...
            then (fnew[key->ad], joined[(adn,adm)->ad], H_new#[ad->on])
            -- the last case is when the two values have different types altogether
            else (fnew[key->ad], joined[(adn,adm)->ad], H_new#[ad->Top_Val])

    in foldl(helper, (fn_emtpy, joined, H_new#), PreIm(fn) U PreIm(fm))
```

`joined` is a function that stores the similarities between nodes (values), and `Hnew#` stores the values of each new value. The type of `Hnew#` is not `Heap#` but `Addr → Val# + Undefined + InConstruction` (the output of `Heap#` extended with an `InConstruction` new value). The new value to which a heap can point to is `InConstruction` and it is meant to be a wildcard value while the graph is being constructed.

One last function, that was not explained before is `removeallInConstruction`. It is in charge of looking in the Heap if there is one reference to some `InConstruction` value left. All `InConstruction` values left after returning from the whole walk will be only those which appeared when joining some node to a `ImBot` or `Bot_Val`. Because we expect `ImBot U fn = fn`, we must copy all values from the joining heaps into the new heap.

Because the "code" describing how to *join* to abstract values can be rather coarsed, we present to you the algorithm working in a couple of examples. The two graphs at the left represent the two values to join, on the right the resulting new value is defined.

$Val^{\#}$ Order definition

Now that we have defined the *join* function, we can move to the function that defines a lattice a lattice.

We define $\leq$ as $a \leq b \iff b = a \sqcup b$. This definition follows from the definition of *join* (based on the order): given any two values $a, b \in$ lattice, it is always possible to find a value that is bigger than a and b such that it is the smallest of the bigger values. In the case $a < b$, then we can be certain that b will be the *join* of a and b, otherwise there would be a value smaller than b that is also bigger or equal than b. That means, that we can define a lattice by either defining an order relation or a proper *join* operation. We leave the proof that $\leq$ defines an order for valid values (`v \in Addr x Heap#`) for future work.

Other definitions

My goal is not to present in detail all functions and operations necessary to define the AD for `Val#s`. That would require three times the space already used in this document, and would be tedious. What I wanted to show here is the rationale behind the implementation to give it a little more formal depth.

The *merge* operation is defined as it was *join*. The main difference between the two is what is the result of operating two values with different types. For *join* different types give us `Top` and for *merge* we get `Bottom`.

A.2.2 State Abstract Domain

Now that we have a Value Abstract Domain, we can extend it to the State of a Program. Doing so it's surprisingly easy, as all the blocks have already been laid down by the Value Abstract Domain.

The State Abstract Domain is defined as the tuple `(Global#, Heap#)` where:

```
-- Global
Global# = Iden -> Addr + Undefined  -- Abstract Global Scope
```

```
-- Heap
Heap# = Addr -> Val# + Undefined    -- Abstract Heap
```

Notice that `Global#` is a function that takes an identifier and outputs an address to where the variable is stored, which is just the same thing that the function `Key→ Addr + Undefined` inside `Object#` does. In fact, the function inside `Object#` has a bigger pre-image than that of `Global#` (`Key` is defined as `Iden + (string x (Iden + PrimVal))`). This means that we have no need to define a *join* operation from scratch for the State Abstract Domain but we just borrow repurpose `joinObjectFun`!

```
UState: (Global#, Heap#) x (Global#, Heap#) -> (Global#, Heap#)
(G1#, H1#) UState (G2#, H2#) :=
  let (Gnew#, joined, Hnew#) = joinObjectFn((G1#, H1#), (G2#, H2#), joined_empty, H_empty)
    in (Gnew#, Hnew#)
```

Similarly all other functions and operations can be defined as a special case of the Value Abstract Domain.

A.2.3 Abstract Semantics

In this section, I present some examples of the abstract semantics. Notice that we do not need the abstraction and concretatisation functions as the State and State Abstract Domains are quite similar. It is left for future work to proof that the definitions stated in here are in fact derived from the Galois connection.

As examples, consider the semantics of expressions:

```
E#[expr] :: Global# x Heap#
         -> Global#
          x Heap#
          x (Val# + Addr + (Object# x (string x Val#)) + Iden)

E#[Name(id, ctx)](G#, H#) :=
  match ctx in
    case Load  -> if G#(id) = Undefined
                -- the variable `id` has not been defined then we "set" it in the global
                  -- scope and return `Top_Val`
                  then let ad := freeaddr(H#)
                        in (G#[id->ad], H#[ad->Top_Val], Top_Val)
                  else (G#, H#, G#(id))
    case Store -> (G, H, id)
    case Del   -> (G, H, id)

E#[BinOp(op, a, b)](G#, H#) :=
  let (G1#, H1#, v1) := E#[a](G#, H#)
      (G2#, H2#, v2) := E#[b](G1#, H1#)
      -- TODO: careful with the two notations you are using to make sure a var is Val#
      --        kind(v) = Val#  vs  isvalue#(v)
   in if kind(v1) /= Val#  or  kind(v1) /= Val#
      -- Bad parsing! `E[e](G, H)` for e in {a,b} are supposed to return Val#
      -- Parsing errors halt the execution of the Abstract Interpreter
      then <Execution Halt>
      else
        let prim_op# := get_prim_op#(op, type(v1), type(v2))
         in prim_op#(v1, v2, G2#, H2#)

TypeVals = Type U {Int, Float, Bool, NoneType}
get_prim_op# :: Op x TypeVals x TypeVals -> <prim-callable>#

get_prim_op#(Add, t1, t2) :=
  match (type(v1), type(v2)) in
    case (Int, Int) -> <prim-+-int>#
    case (Float, Float) -> <prim-+-float>#
    case (Int, Bool) -> \(i, j, G#, H#)-> <prim-+-int>#(i, Int(j), G#, H#)
    case (Bool, Int) -> \(i, j, G#, H#)-> <prim-+-int>#(Int(i), j, G#, H#)
    case (Float, a) -> if a = Bool or a = Int
                       then \(i, j, G#, H#)-> <prim-+-float>#(i, Float(j), G#, H#)
                       else <prim-ret-top>#
    case (a, Float) -> if a = Bool or a = Int
                       then \(i, j, G#, H#)-> <prim-+-float>#(Float(i), j, G#, H#)
                       else <prim-ret-top>#
    -- This function is to be extended once we add NdArrays to the mix
    case otherwise -> <prim-op-top>#

<prim-op-top># : Val# x Val# x Global# x Heap#
<prim-op-top>#(a, b, G#, H#) := (G#, H#, Top_Val)
```

Notice that parsing errors still halt the execution of the Abstract Interpreter. It could be possible to work around those errors too, but parsing errors out of the scope of the Abstract Interpreter as they hint to an external problem. We assume that a piece of code is parsed properly before starting the execution of the code (just as Python does).

Every function defined in the formal semantics of Python must be rewritten as function that operates in the Abstract Domain. For example, consider the function `<prim-+-int>`, it is defined as:

`<prim-+-int>(i, j, G, H) := (G, H, i+j)`

We need to define its Abstract Interpretation counterpart:

```
<prim-+-int>#(i, j, G#, H#) :=
  if i = Top_Int or j = Top_Int
  then (G#, H#, Top_Int)
  else if i = Bot_Int or j = Bot_Int
  then (G#, H#, Bot_Int)
  else (G#, H#, i+j)
```

And as an example for the semantics of statements consider:

```
S#[Assign(var, val)](G#, H#) :=
  let (G1#, H1#, ass) := E[var](G#, H#)
      (G2#, H2#, rightval) := E[val](G1#, H1#)
     -- Parsing error, probably. `rightval` must be a `Val#` if the parsing made no mistake
```

```
      rval := if is_value#(rightval) then val else <Execution Halt>
   in
      match ass in
        case Iden -> (G2#[ass->rval], H2#)

        case ((t, addr, o): Object#, ('index', val: PrimVal#)) ->
          let setindex# := get_prim_set_index#(t)
          in setindex#(G2#, H2#, o, val, addr, rval)

        -- Parsing error probably. This should never have happened
        otherwise -> <Execution Halt>

S#[Import(name)](G#, H#) :=
  let freeaddr := get_free_addr(H#)
   in
     match name in
       ("numpy",) ->
         let (Module, arr, mod) := <numpy-mod>
         in  (G#['numpy'->freeaddr], H#[freeaddr->(Module, freeaddr, mod)])
       ("numpy", alias) ->
         let (Module, arr, mod) := <numpy-mod>
         in  (G#[alias->freeaddr], H#[freeaddr->(Module, freeaddr, mod)])

       ("pytropos.hints.numpy",) ->
          (G#["pytropos.hints.numpy"->freeaddr], H#[freeaddr->(Module, freeaddr, <numpy-
hints>)])
       ("pytropos.hints.numpy", alias) ->
          (G#[alias->freeaddr], H#[freeaddr->(Module, freeaddr, <numpy-hints>)])

       (nm,) ->
          (G#[nm->freeaddr], H#[freeaddr->(Module, freeaddr, ImTop)])

       (nm,alias) ->
          (G#[alias->freeaddr], H#[freeaddr->(Module, freeaddr, ImTop)])
```

A.2.4 Type Annotations

I left out, purposefully, the abstract semantics of `AnnAssign`. The idea of type annotations is that they let us refine the value/type of a variable when the Abstract Interpreter is unable to define a precise value.

An example of what annotations should be able to do is:

```
from mypreciouslib import amatrix
from pytropos.hints.numpy import NdArray

mat: NdArray[1,12,6,7] = np.array(amatrix)
newmat = (mat + 12).reshape((12*6, -1))  # newmat has shape (12*6,7)
newmat = newmat.dot(np.ones((8, 21)))  # Error! Matrix multiplication, 7≠8
```

When one imports an arbitrary library like `mypreciouslib` it is impossible to know what we may have imported so every variable imported is a `Top_Val`, i.e. anything. By adding an annotation to `mat`, we are telling the abstract interpreter that we know, and are sure, of the shape of the NumPy array.

The semantics for `AnnAssign` are:

```
S[AnnAssign(var, hint, val)](G, H) :=
  let (G1, H1, evaluatedhint) := E[hint](G, H)
      (G2, H2, evaluatedhint) := E[hint](G1, H1)
      hintval := if isvalue#(evaluatedhint) then evaluatedhint else <Execution Halt>

   in S[Assign(var, val)](G1, H1)

  let (G1, H1, ass) := E[var](G, H)
      (G2, H2, evaledhint) := E[hint](G1, H1)
      (G3, H3, rightval) := E[val](G2, H2)
      compval := if is_value(rightval) then val else <Execution Halt>
      hintval := if isvalue#(evaledhint) then evaledhint else <Execution Halt>

      -- if `hintval` is more precise than `compval` we replace it
      rval := if hintval < compval then hintval else compval
   in
      ... -- Continue as in S[Assign( ... )]
```

Notice that we determine the "precision" of a hint with respect to the variable with the comparision between values, the order `<Val#`!

Appendix B

Gotchas and Other ideas to Static Analysis in Python

Note: This chapter is probably the most informal of the whole book. The principal idea of this chapter is to explain a little bit deeper the history of the whole thesis, some of the things I tried to solve the problem at hand, and what avenues I found could be explored and which are probably dead ends. This whole chapter is me rambling about various stuff

In here, I write on the multiple other ways I thought on how to Statically analyse the shape of tensors. I explored some of them but never really finish them, while others just proved to be bad ideas altogether.

Some background. The principal, original, idea for my master thesis was to check the shape of tensors and check the validity of tensor operations. I did not consider Abstract Interpretation as my first option to statically analyse code. In fact my whole exposure to Static Analysis had been Type Inference, some Data Flow Analysis algorithms, and some exposure into SMT solvers theory and practice.

I will mention each one of the approaches I tried in chronological order.

Tensor Shape Type Checking - Haskell

Everything started in Haskell. I wrote a wrapper library around a Haskell wrapper library for TensorFlow (Abadi et al., 2016). The library uses one of the lastest additions to the language, Dependent Types (Eisenberg, 2016). Dependent Types allow me to define the shape of tensors at the type level, which allows the compiler to check for the correctness of tensor operations at compilation time.

My main motivation to use Haskell is its very strict type system, and the tools that it incorporates to work with types. For example, in GHC, the standard Haskell implementation, one can ask for the inferred type of an expression by annotating the type with the type symbol `_`[1], thus we can ask what shape of tensor should we use in the middle of an operation.

Although, type checking tensor shapes in Haskell proved to be possible and not all too hard, albeit a very verbose at times, I yet have to know the first Deep Learning enthusiast who writes in Haskell. So, I decided to apply the same idea into Python, the "language" of Deep Learning.

Some efforts have been made to add type inference and type checking into Python (See Related Work, Chapter 6)), the most important one being MyPy.

Extending MyPy

MyPy builds on the idea of Gradual Typing. MyPy is able to infer the type of primitive values and allows optional annotations to check the code. Sadly, MyPy does not consider literal numbers as valid types (`int`, `float`, `List[int]` are valid types but no `3` or `List[2, int]`), so it is currently impossible to type annotate a variable with the size of the variable, e.g. something like `np.ndarray[2, 3, 4]`.

[1]They are called holes. There are two types of holes: (regular) holes and type holes. A hole is an unknown expression and a type hole is an unknown type. You can put a whole at the type level and the compiler will tell you what is the type that should be at that position. I found this incredibly neat and useful!

But hope is still not lost! We can encode numbers in MyPy's Type System in a similar way to how was done in (Chen, 2017; Eaton, 2006). We can encode 1 as `Tuple[None]`, 2 as `Tuple[None, None]`, and so on.

Additionally, we can compare fields between two variables in an operation with the use of `TypeVars` (variables at the type level). With these two ideas, we can rudimentary check the shapes of tensors. First we write a stub file (Guido van Rossum et al., 2014) with type annotations to check for the shape of NumPy arrays:

```
# numpy.pyi
from typing import Any, Generic, Optional, Tuple, TypeVar

S = TypeVar('S')  # Used for dtype
Shape = TypeVar('Shape')  # Used for the shape
s1 = TypeVar('s1')  # Another variable to use as reference to a shape
s2 = TypeVar('s2')
s3 = TypeVar('s3')


class ndarray(Generic[S, Shape]):
    def __init__(self, shape: Any) -> None:  ...

    def dot(
            self: 'ndarray[S,Tuple[s1,s2]]',
            b: 'ndarray[S,Tuple[s2,s3]]'
    ) -> 'ndarray[S,Tuple[s1,s3]]':  ...

    def __add__(
            self: 'ndarray[S,Shape]',
            value: 'ndarray[S,Shape]'
    ) -> 'ndarray[S,Shape]':  ...


def array(object: Any) -> ndarray[Any, Shape]:  ...
```

And now we write the piece of code to type check:

```
import numpy as np
from typing import Tuple

one   = Tuple[None]
two   = Tuple[None, None]
three = Tuple[None, None, None]
four  = Tuple[None, None, None, None]

# x has shape (2,3)
x = np.array([[1, 2, 3], [4, 5, 6]])  # type: np.ndarray[float, Tuple[two, three]]

# y has shape (4,1)
y = np.array([[7], [0], [2], [1]])    # type: np.ndarray[float, Tuple[four, one]]
```

```
w = x + y  # fails! And MyPy warns us!! :D

# THIS SHOULD FAIL IN MYPY!! But it doesn't :(
z = x.dot(y)  # type: np.ndarray[float, Tuple[two, one]]
```

Everything seems to work perfectly, MyPy detects that the two numpy arrays cannot be added together because the have different shapes, but MyPy does not detect the error on multiplying two non-compatible matrices. We told MyPy in the stub that the shape of the last dimension in `self` (`s2`) should be the same as the first dimension in `b`, but MyPy does not verify this condition (`three ≠ four`).

Unfortunatelly, MyPy's `TypeVars` are not fully-fledged type variables. MyPy does not handle Dependent Types, so a solution as the one suggested for Haskell could not completely be coded.

Two options came from this experiment, either look how to extend MyPy type variables handling to work in the general case or ignore type checking as a possible solution at the moment.

StyPy

As I mentioned in Chapter 4, before I encountered Abstract Interpretation as a way to Statically Analyse code[2] I found *StyPy* (Ortin et al., 2015).

The idea behind StyPy is to reuse the infraestructure of the language to analyse to not write everything from scratch. The idea is to take the code to analyse and rewrite/transform it into a new piece of code that when run it performs Static Type Analysis. This approach presents to difficulties: first, it was planned as a technique for Static Type Analysis not Static Value Analysis, and second, it is a very naïve and novel idea, it is not very well formalised.

When you require to know only the type of a variable, its value becomes irrelevant. If we do not consider any weird introspection Python capabilities, it does not matter how many times you run a loop a variable will always keep the same type after the first loop iteration, i.e. there is no need to consider complex scenarios and joining operations for the type of a variable as we need to consider for the value of a variable inside a loop.

There have not been any updates on the theory behind the project since the paper was published in 2015.

All in all, the idea of reusing some of the infraestructure by rewriting the code and running it in the same platform where one wants to analyse the code is a very interesting idea. In fact, this idea has been applied in other static analyses, for example, Lauko et al. (2018) explain how they built a symbolic analyser for LLVM bytecode by reusing some of the infraestructure of LLVM and not rewriting a complete interpreter for LLVM from the ground up.

Pytropos started from the same idea of StyPy, transform and run modified code. The idea would be that any variable in the (extended) interpreter could be any of Python's or `Any` (`?`, the special type introduced in Gradual Typing). An `Any` value would tell me that I have no idea of the value contained in the variable, just that the variable contains a variable. I discarded the idea of modifying StyPy as it would mean rewriting the whole program for my purpose. So I went on looking different projects that I could modify to implement the Static Value Analysis I was considering.

Modify PyPy

[2] I know, it is a very old, well understood, and broadly applied analysis technique but I didn't know much about the broad world of Static Analysis before I delved deep into it.

Once I considered building a special purpose interpreter to analyse the shape of tensors (as an Abstract Interpreter is), I took a look in other projects that had implemented interpreters for python. a very interesting project is pypy. pypy (Holger Krekel et al., 2007) is a project focused on providing an alternative implementation of python with added characteristics. most importantly, pypy is written in python (or a subset of python, called rpython) and pypy can be extended/its semantics can be modified. apparently, pypy offers a way to change the semantics of the interpreter[3].

both characteristics; pypy being written in python, allowing for an easy introspection and modification of its internals, and the ease of modifing some of its semantics; made me think of using pypy as a "base" for the custom made interpreter. i did not follow this route because the work to be done to allow having different states of a program and joining them (used in `if` branching when choice can be made between two branches) seemed to require rewritting a big chunk of the core of pypy. starting from zero seemed a better idea.

abstract interpretation

in the end, i decided to build my own interpreter from scratch. it took me a couple of months to arrive to something relatively usable but i found out in the process that branching, loops, exceptions, and breaking control statements made the implementation of a interpreter that used `any` values very challenging (how are the semantics of a non-deterministic `if`, of a loop, an exception, etc).

it became clear that trying to build an (extended) interpreter from scratch would be very hard if no clear formal framework was used to put everything into perspective. enter abstract interpretation. i cannot say that i found abstract interpretation as the solution for the many problems that an extended value "system" presented, because i had no idea such thing existed.

as you may have noticed, i just arrived a the "proposed" solution of my problem (static value analysis) by "mere" chance. i was looking at the literature of static analysis, but i thought there was little i could use from it. how wrong was i! always so smartass.

anyway, once i studied together with my advisors the theory of abstract interpretation, i knew that ai was the solution for my problem. it was a long path nontheless, as it required to formalise a couple of things about the language and define an abstract domain for the state of the language, which proved to be challenging but rewarding.

now, i must mention here that i did came across a project that was doing pretty much what i wanted to do, namely an abstract interpreter for python. lyra is a project led by which purpose is to build an abstract interpreter for python. there are two reasons why i chosed not to use lyra for this thesis: the first is purely egothistical and not practical at all, i wanted to build something from my own, something from "scratch"[4]; the second reason was that i had found the idea of reusing some of the python infraestructure too fascinating to let it into oblivion, rewriting python code to run it into with python seems just a nice metaidea.

Other Ideas

Other routes, I did considered to follow but did not because of time restrictions where:

- If all we wanted was to check the shape of tensors and tensor operations for a couple of scenarios, we could build a Static Analysis out of various Data Flow Algorithms. Data flow analyses are simple, well studied, and already implemented in a couple of Static Analysis frameworks . The only work would be combining them adecuately to know the shape of tensors and consequently check for the correctness of a couple of cases.

[3]the object space can be modified as explained in http://doc.pypy.org/en/latest/objspace.html

[4]please, dear advisors, do not kill me for writing this!

- Another option may have been encoding the shapes as constraints. The constraints could be checked using an out of the shelve equation solver (SMTs). In fact, a big chunk of the work for this has already been done, an implementation for Python for example .

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., ... Zheng, X. (2016). TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *arXiv:1603.04467 [cs]*. arXiv: 1603.04467. Retrieved October 11, 2017, from http://arxiv.org/abs/1603.04467

Abe, A., & Sumii, E. (2015). A Simple and Practical Linear Algebra Library Interface with Static Size Checking. doi:10.4204/EPTCS.198.1

An, J.-h. (, Chaudhuri, A., Foster, J. S., & Hicks, M. (2011). Dynamic Inference of Static Types for Ruby. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (pp. 459–472). POPL '11. doi:10.1145/1926385.1926437

Arnold, G., Hölzl, J., Köksal, A. S., Bodík, R., & Sagiv, M. (2010). Specifying and Verifying Sparse Matrix Codes. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (pp. 249–260). ICFP '10. doi:10.1145/1863543.1863581

Bergstra, J., Bastien, F., Breuleux, O., Lamblin, P., Pascanu, R., Delalleau, O., ... Bergeron, A., et al. (2011). Theano: Deep learning on gpus with python. In *Nips 2011, biglearning workshop, granada, spain* (Vol. 3). Citeseer.

Bierman, G., Abadi, M., & Torgersen, M. (2014). Understanding typescript. In *European conference on object-oriented programming* (pp. 257–281). Springer.

Chaudhuri, A. (2016). Flow: Abstract interpretation of javascript for type checking and beyond. In *Proceedings of the 2016 acm workshop on programming languages and analysis for security* (pp. 1–1). ACM.

Chen, T. (2017). Typesafe Abstractions for Tensor Operations. *arXiv:1710.06892 [cs]*, 45–50. arXiv: 1710.06892. doi:10.1145/3136000.3136001

Cousot, P., & Cousot, R. (1977). Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (pp. 238–252). POPL '77. doi:10.1145/512950.512973

Cruz-Camacho, E., & Bowen, J. (2018). Tensorflow-haskell-deptyped: Reexporting tensorflow haskell with dependent typed functions. https://github.com/helq/tensorflow-haskell-deptyped. GitHub.

Eaton, F. (2006). Statically typed linear algebra in Haskell. In *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell* (pp. 120–121). ACM.

Eisenberg, R. A. (2016). Dependent types in Haskell: Theory and practice. *arXiv preprint arXiv:1610.07978*. Retrieved July 18, 2017, from https://arxiv.org/abs/1610.07978

Foundation, P. S. (2019). The Python Language Reference — Python 3.6.8 documentation. Retrieved February 12, 2019, from https://docs.python.org/3.6/reference/

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* (pp. 185–202). Lecture Notes in Computer Science. Springer International Publishing.

Griffioen, P. R. (2015). Type Inference for Array Programming with Dimensioned Vector Spaces. In *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages* (4:1–4:12). IFL '15. doi:10.1145/2897336.2897341

Guido van Rossum, & Ivan Levkivskyi. (2014). PEP 483 – The Theory of Type Hints. Retrieved February 27, 2019, from https://www.python.org/dev/peps/pep-0483/

Guido van Rossum, Jukka Lehtosalo, & Łukasz Langa. (2014). PEP 484 – Type Hints. Retrieved February 27, 2019, from https://www.python.org/dev/peps/pep-0484/

Guth, D. (2013). A formal semantics of Python 3.3.

Hejlsberg, A. (2012). Introducing typescript. *Microsoft Channel*, *9*.

Holger Krekel, Lene Wagner, Jacob Hallén, Beatrice During, Carl Friedrich Bolz, Laura Creighton, ... Maciej Fijalkowski. (2007). Ist fp6-004779 pypy - final activity report. Technical report, PyPy Consortium. Retrieved from http://doc.pypy.org/en/latest/index-report.html

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., ... Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*.

Kazerounian, M., Vazou, N., Bourgerie, A., Foster, J. S., & Torlak, E. (2017). Refinement Types for Ruby. *arXiv:1711.09281 [cs]*. arXiv: 1711.09281. Retrieved December 2, 2017, from http://arxiv.org/abs/1711.09281

Lauko, H., Ročkai, P., & Barnat, J. (2018). Symbolic Computation via Program Transformation. *arXiv:1806.03959 [cs]*. arXiv: 1806.03959. Retrieved June 26, 2018, from http://arxiv.org/abs/1806.03959

Lehtosalo, J. et al. (2016). Mypy (2016). Retrieved from http://mypy-lang.org/

Miné, A. (2004). *Weakly relational numerical abstract domains* (PhD Thesis, Ecole Polytechnique X).

Monat, R. (2018). *Static Analysis by Abstract Interpretation Collecting Types of Python Programs*. LIP6 - Laboratoire d'Informatique de Paris 6. Retrieved February 12, 2019, from https://hal.archives-ouvertes.fr/hal-01869049

Nielson, F., Nielson, H. R., & Hankin, C. (2015). *Principles of program analysis*. Springer.

Nipkow, T., & Klein, G. (2014). Abstract Interpretation. In T. Nipkow & G. Klein (Eds.), *Concrete Semantics: With Isabelle/HOL* (pp. 219–280). doi:10.1007/978-3-319-10542-0_13

Oliphant, T. E. (2006). *A guide to numpy*. Trelgol Publishing USA. Retrieved from http://www.numpy.org/

Ortin, F., Perez-Schofield, J. B. G., & Redondo, J. M. (2015). Towards a static type checker for python. In *European Conference on Object-Oriented Programming (ECOOP), Scripts to Programs Workshop, STOP* (Vol. 15, pp. 1–2).

Paszke, A., Gross, S., Chintala, S., & Chanan, G. (2017). Pytorch.

Pierce, B. C., & Benjamin, C. (2002). *Types and programming languages*. MIT press.

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

Rakić, P. S., Stričević, L., & Rakić, Z. S. (2012). Statically typed matrix: In C++ library. In *Proceedings of the Fifth Balkan Conference in Informatics* (pp. 217–222). ACM.

Ranson, J. F., Hamilton, H. J., Fong, P. W., Hamilton, H. J., & Fong, P. W. L. (2008). A Semantics of Python in Isabelle/HOL.

Rink, N. A. (2018). Modeling of languages for tensor manipulation. *arXiv:1801.08771 [cs]*. arXiv: 1801.08771. Retrieved January 29, 2018, from http://arxiv.org/abs/1801.08771

Rushby, J., Owre, S., & Shankar, N. (1998). Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, *24*(9), 709–720.

Ryan Gonzalez, Philip House, Ivan Levkivskyi, Lisa Roach, & Guido van Rossum. (2016). PEP 526 – Syntax for Variable Annotations. Retrieved February 27, 2019, from https://www.python.org/dev/peps/pep-0526/

Siek, J. G., & Taha, W. (2006). Gradual typing for functional languages. In *Scheme and Functional Programming Workshop* (Vol. 6, pp. 81–92).

Slepak, J., Shivers, O., & Manolios, P. (2014). An Array-Oriented Language with Static Rank Polymorphism. In *Programming Languages and Systems* (pp. 27–46). Lecture Notes in Computer Science. doi:10.1007/978-3-642-54833-8_3

Trojahner, K., & Grelck, C. (2009). Dependently typed array programs don't go wrong. *The Journal of Logic and Algebraic Programming*. The 19th Nordic Workshop on Programming Theory (NWPT 2007), *78*(7), 643–664. doi:10.1016/j.jlap.2009.03.002

Urban, C. (2015). *Static analysis by abstract interpretation of functional temporal properties of programs* (Doctoral dissertation, Ecole normale supérieure - ENS PARIS). Retrieved July 25, 2018, from https://tel.archives-ouvertes.fr/tel-01176641/document