



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

Elkin Alejandro CRUZ CAMACHO

Universidad Nacional de Colombia
Facultad de Ingeniería
Departamento de Ingeniería de Sistemas e Industrial
Bogotá, Colombia
2019

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

Elkin Alejandro CRUZ CAMACHO

Tesis presentada como requisito parcial para optar por el título:
Magíster en Ingeniería de Sistemas y Computación

Director:

PhD. Felipe RESTREPO CALLE

Co-director:

PhD. Fabio Augusto GONZÁLEZ

Línea de Investigación:

Lenguajes de Programación

Grupos de Investigación:

PLaS - Mindlab

Universidad Nacional de Colombia
Facultad de Ingeniería
Departamento de Ingeniería de Sistemas e Industrial
Bogotá, Colombia
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*

*A mi mamá y a mi abuelita
¡Gracias por todo lo que han hecho por mí!*

Acknowledgements

Resumen

Los tensores, una extensión de los arrays, se usan de manera extensiva en una gran variedad de problemas en programación. Los tensores son los bloques básicos de construcción de múltiples frameworks de Aprendizaje de Máquina y son fundamentales en la definición de modelos de Aprendizaje Profundo. Los linters son herramientas indispensables para los programadores de hoy en día, ya que estos ayudan a los desarrolladores a revisar el código antes de ejecutarlo. Aunque los tensores son muy populares no existen tensores que revisen código con operaciones tensoriales. Dada la gran cantidad de trabajo hecho en Python con (y sin) tensores, es sorprendente el poco trabajo que se ha hecho en esta área. La Interpretación Abstracta es una metodología/framework diseñada para analizar código de forma estática. La idea de Interpretación Abstracta es sobreaproximar de manera "sound" el resultado de ejecutar una pieza de código sobre todos las posibles entradas del programa. Una sobreaproximación "sound" asegura que el Interprete Abstracto nunca omitirá un verdadero negativo, es decir, si una pieza de código no es señalada como incorrecta por el Interprete Abstracto entonces se puede asumir con seguridad que el código nunca fallará. El Interprete Abstracto puede ser modificado para que sólo informe acerca de verdaderos positivos, aunque se pierda la propiedad de "soundness", es decir, el interprete sólo informa acerca de las partes de código que fallarán sin importar que suceda.

En este trabajo, formalizamos un subconjunto de Python con énfasis en operaciones con tensores. Nuestra formalización de la semántica de Python está basada en la Referencia oficial del Language Python. Definimos un Interprete Abstracto y presentamos su implementación. Mostramos como cada parte del Interprete Abstracto fué definido: su Dominio Abstracto y semántica abstracta.

Presentamos la estructura de Pytropos, la implementación del Interprete Abstracto. Pytropos es capaz de revisar las operaciones de arreglos de NumPy teniendo en cuenta broadcasting y algunas funciones complejas de NumPy como `array` y `dot`. Construimos 74 casos de prueba unitarios, los cuales chequean la capacidad de Pytropos, además de 20 casos de prueba de propiedades, los cuales chequean que las reglas semánticas de Pytropos correspondan con la forma en la que Python es ejecutado. Mostramos las capacidades del Interprete Abstracto por medio de ejemplos de los test unitarios.

Abstract

Tensors, an extension of arrays, are widely used in a variety of programming tasks. Tensors are the building blocks 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 specify a subset of Python with emphasis on tensor operations. Our operational Python semantics is based on The Python Language Reference. We define an Abstract Interpreter and present its implementation. We show how each part of the Abstract Interpreter was built: the Abstract Domains defined and the abstract semantics.

We present the structure of Pytropos, the Abstract Interpreter implemented. Pytropos is able to check NumPy array operations taking into account broadcasting and complex NumPy functions as `array` and `dot`. We constructed 74 unit test cases checking the capabilities of Pytropos and 20 property test cases checking compliance with the official Python implementation. We show what and how many bugs the Abstract Interpreter was able to find.

Research, the verb, is as any other skill, a thing that you develop only with time and practice. I would lie if I did not say that the last two years have felt like an intensive camp to learn to research. Research is hard, but it is harder to try to explain what you have done to others, but as I discovered writing this document and taking notes, writing what you think it is one of the most useful and important parts of the *journey*. Research is not always fun, failing at it is depressing, and as if it was not enough, research is often boring and arduous. Fortunately, I was not alone in this journey, I have been surrounded by masters in this art and I would like to thank them.

To my advisors:

- Felipe Restrepo, who has been very supportive and who has held wave after wave of my bad written texts.
- Fabio González, who has been flexible enough to let me explore different areas on my own. Without his critics on my wild ideas, I would not have put the effort to show that some of those ideas actually had some substance.

To the staff and people at the university:

- Jonatan Gómez, who told me with years of anticipation what I should not be doing. His knowledge in what could be wrong has always been an inspiration to do better.
- Rodrigo Moreno, Oscar Perdomo, and all the others in the Mindlab group for their support and company in the long and arduous days at the university.
- To all my students, who showed me how hard it is to actually program human minds. Some, very susceptible to change and others impossible to do so.

No human being can live without others. We all need a social net, something to support ourselves. I would like to acknowledge their time on keeping me distracted from insanity.

To my friends:

- Sergio Dorado, who motivated me to do my best and find new goals. He contagated me—I envy him—with his ideas and accomplishments.
- Helen Smith, who, even 8 thousand kilometres away, was able to find time to chat with this Colombian guy.
- William, Solene and Vinicius for their time and memories.
- Vicky, whose perseverance I envy, and whose work and learning ethic I try to imitate.

To the external events that delayed or accelerated this thesis culmination:

- To the “paro,” a students’ university strike, for giving me 3 extra months to write this document, and
- To the sicknesses of my family members, which helped me centre myself in what is important.

I could go on and on. There are hundreds of people and little pieces of reality that made me who I am, and who helped me to get to the end. I am grateful of having them all, albeit my faulty memory is not able to remember them all. Sorry you guys.

But certainly, I could have never done a thing without the help of my family:

- Isabel Camacho, my mom, who has done all the boring tasks to let me concentrate on the work, I owe her my life (and not only literally).
- María del Carmen Gordillo, my granma, who has always been keen on my and my mom’s health.

- My cousins Paola, Laura, Danilo, Angie, Duvan, and Karol who have always given me some excuse to relax, and from whom I have learnt so much about the world.
- My aunties Marta, Teresa and Patricia, and my Uncle Juan for their support.
- My dad, without whom I would not be here.

To all, thank you for being there.

Contents

Acknowledgements	vii
Resumen	ix
Abstract	xi
Prelude	1
1 Introduction	3
1.1 Motivation	3
1.2 Problem Definition	4
1.3 Objectives	7
1.3.1 General objective	7
1.3.2 Specific objectives	8
1.4 Contributions	8
1.5 Thesis Structure	8
2 Background	9
2.1 Dynamic and Static Analysis	9
2.2 Python Type Annotations	10
2.3 Machine Learning and Python	10
2.4 Tensors	11
2.5 NumPy: Library for tensor computation	11
2.6 Abstract Interpretation	12
2.6.1 Informal Introduction	12
2.6.2 Ingredients for an Abstract Interpreter	13
2.6.3 Value Abstract Domain	14
2.6.4 State Abstract Domain	17
2.6.5 Abstract Semantics	19
2.6.6 Missing Bits	19
3 Abstract Interpretation for Python	21
3.1 Semantics	21
3.2 Concrete Semantics of Python	22
3.3 Abstract Semantics	26
3.3.1 State Abstract Domain	26
3.3.2 Semantics for the Abstract State	30
3.4 NdArrays	31
3.5 Type Annotations	33

4	Pytropos (Analysis Implementation)	35
4.1	The big picture	35
4.2	Wrapped code + libs + interpreter vs. from scratch interpreter	36
4.3	Assumptions	37
4.4	Details about the guts	38
5	Validation and Discussion	41
5.1	Validation	41
5.1.1	Unit Tests	41
5.1.2	Property-based Tests	42
5.2	Discussion	42
5.2.1	Pytropos capabilities	43
5.2.2	Pytropos failures	46
5.2.3	Using Pytropos as a linter	47
5.2.4	Summary	48
6	Related Work	49
6.1	Static Analysis in Python	49
6.2	Tensor Shape Analysis	49
6.2.1	Solutions in other languages	49
6.2.2	Theoretical Solutions	51
7	Conclusions and Future Work	53
7.1	Conclusions	53
7.2	Future Work	54
A	Python Static Analysis based on Abstract Interpretation	55
A.1	Python (reduced) syntax and semantics	55
A.1.1	(Reduced) Syntax	56
A.1.2	Python (reduced) small step semantics	58
A.1.3	NdArrays	66
A.2	Abstract Interpreter	68
A.2.1	Variable Abstract Domain	68
A.2.2	State Abstract Domain	76
A.2.3	Abstract Semantics	77
	Bibliography	81

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 (specifically, to my own one-and-a-half-years past self). 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 (dynamically typed languages for short) like Python and Ruby have risen in popularity over the last two decades. Their simple syntax, ease of use, out-of-the-box REPLs¹, dynamic type systems, and their great number of libraries have been the main factors to this rise. Indeed, developers can write, execute and see results immediately in fast development cycles, especially at the initial stages. However, their simple syntax and dynamic typing make them particularly difficult to analyse as the more rigid the language, the easier to analyse it is. Most efforts on static analysis for Python have focused on assuming some restrictions on the language. For example, MyPy (Lehtosalo et al., 2016) assumes the type of variables as static.

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

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

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

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. it detects in the piece of code from above that `other` is of type `float` and `myvar` of type `str`.

Python dynamic nature makes it extraordinarily hard to statically analyse. In fact, due to Python introspection capabilities, it is impossible to guarantee the behaviour of a piece of Python code without assuming some fixed basic semantics 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 system².

¹Read-eval-print loop

²I would like to express my immense 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. All Python code I write now is full of typing annotations to allow MyPy to check for any inconsistency. What I like about MyPy is that it alerts me of faulty code as I write. It feels remarkably similar to writing code in a language with a stronger static type system (e.g. Haskell).

The inference power of MyPy is limited in several ways, one of them being it impossible to infer the type of a function. MyPy allows developers to optionally add type annotations into the code. Type annotations allow more restrictive and precise types, and they help MyPy to catch more potential bugs.

Although MyPy is capable of checking for type mismatches, its assumptions limit it from a whole class of valid and correct Python code³. 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:

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

In this example, we are using NumPy (Oliphant, 2006), which is a well known Python library, but it could also have been TensorFlow (Abadi et al., 2016) or any other library with support for tensors and tensor operators. Tensors are a generalisation of vectors and arrays, a homogeneous container which can be indexed by one natural number or a list of natural numbers. As an example consider a tensor of only one dimension, an array of size n , which contains exactly n elements. A matrix of size $n \times m$, alternatively, a tensor with shape (n, m) , is a structure that holds $n \cdot m$ elements. Notice that a shape is any tuple of natural numbers and a tensor is a structure with a shape. For example, the shape $(2, 5, 6)$ tells us that the tensor has $2 \cdot 5 \cdot 6 = 60$ elements.

The example above will fail to run 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 the developer will not be warned about it and will only notice it when executing the code. To summarise, the aforementioned piece of code 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 shapes. Unfortunately, there does not seem to be a straightforward way to add the shape of tensors to their 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, tensors have become a central figure in the developers' toolkit. Tensor operations fail if the tensors are not of the right shape. To statically analyse if operations between tensors are valid, the shape of tensors must be computed. To compute the shape of tensors, we require to also compute the value of every other variable in the language. The kind of static analysis that is focused on finding out the value of variables is 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 using tensors and tensor operations. A Static Value Analysis tool focused on tensor shape analysis could aid developers in their work.

³For 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 annotation is given, MyPy alerts the user on the usage of non-typed functions. But no type can satisfy the assertion because of the “polymorphism” of the `+` operation.

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

Practical attempts to check for tensor shapes have been mainly focused on languages with strong type systems. 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 all the 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 to follow in order 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 formal semantics for Python have been done (Politz et al., 2013; Fromherz, Ouadjaout, & Miné, 2018; Guth, 2013; Ranson, Hamilton, Fong, Hamilton, & Fong, 2008) but none of them takes into account type annotations, which we hope to integrate into the Static Value Analysis.

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

Assumptions and Scope

In this section, we will explore some of the assumptions and scope of the Abstract Interpreter to be build in the rest of the document.

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:

```
1 # runme.py
2 a = int('6') + 2
3 print(7)
4 assert a == 8, "int('6') + 2 != 8. Wut?"
```

One may be tempted to say that the code above never fails, but one would be mistaken. It will never fail to run if we assume that the program starts from a blank state, i.e. no variables have been defined before its execution. In fact, the functions `int` and `print` can be redefined to compute anything we want. Also, thanks to the introspection capabilities of Python, the code above could do just about anything, and that is without considering that the piece of code could be evaluated by an `exec` or an `eval` statement. For example, consider the following piece of code that calls `runme.py`:

```

1  runme = open('runme.py').read()
2  fakerun = """
3  def int(val):
4      return 3.0
5  def print(val):
6      global a
7      a = val
8  """ + runme
9  exec(fakerun)

```

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 built-in function or variable (this includes the behaviour of variables and functions from non-built-in libraries like NumPy), i.e. we assume that the code will be run from a blank state with no global variables (re)defined.

As a developer, I can say that I like when I see a piece of code flagged wrong and IT IS WRONG and not a false alarm, a false positive. 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 false positives.

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, we chose to focus on checking NumPy's tensor operations. NumPy is the standard library for computing with tensors and it is used as back-end on most tensor 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 just on a couple of Python core characteristics and will leave others for future work. The characteristics explored in this current document are:

- Built-in primitive variables: `int`, `float`, `bool`, `None`, `list`, `tuple`.
- Primitive functions: `int`, `print`, `input`, ...
- Boolean and numeric operations: `+`, `-`, `*`, `/`, `\%`, `**`, `<<`, `>>`, `//`, `<`, `<=`, `>` and `>=`.
- `if` and `while` statements.
- 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:

```

1  if someval:
2      i = 3
3  else:
4      i = "ntr"
5  print(i)

```

The variable `i` can either be an `int` or a `str`, thus the type of `i` should be `Union[int, str]`.

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 the type of a variable 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.

Building an Abstract Interpreter aware of `Union` types is not considered part of this work because of the added complexity it would require. Therefore if a variable holds more than one value, e.g. it is either a `5` or `2.3`, then the type of the variable will be considered to be `Any`, not `Union`. `Any` and `Union` types are defined in the `typing` library (Guido van Rossum, Jukka Lehtosalo, & Łukasz Langa, 2014), an implementation of Gradual Typing⁴.

It may seem like an Abstract Interpreter where variables can only carry one value at the time would make the analysis weaker than Gradual Typing, but it does not make it necessarily. Consider the following piece of code:

```
1 i = 3
2 i = "ntr"
3 i += "ueor"
4 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. The Abstract Interpreter would run each line sequentially, and it would never find a state where 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 abstract interpreter are:

- The behaviour of built-ins, imported variables and functions is always the same, and it 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 of errors that will happen but no errors that *may* happen (This means that the resulting tool is not a verification tool).
- The Static Value Analysis should focus on 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.

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

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 interpreter.
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:

- An specification of operational semantics for a subset of Python using a methodology similar to Fromherz et al. (2018). This specification 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.
- Definition of an Abstract Domain for variables in Python.
- Definition of an aliasing-aware Abstract Domain for the state of a Python program.
- Derivation of an operational semantics from the Abstract Domain, i.e. an Abstract Interpreter for Python.
- A working, open source implementation of the Abstract Interpreter nicknamed Pytropos⁵ with a plugin for (Neo)Vim to work as a linter.

1.5 Thesis Structure

The remaining parts of this document are organised 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 nicknamed Pytropos, with the goal of showcasing the abilities of Pytropos to detect errors 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.
- Additionally, an Appendix A to explain in detail the parts of the abstract interpreter. The appendix 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 some other ways in which Statically Analyse the shape of tensors for Python.

⁵The source code for the interpreter can be found at <https://github.com/helq/pytropos>

Chapter 2

Background

In this chapter, we explore a couple of motivational topics and ideas necessary for the development of the Static Value Analysis presented in Chapter 3.

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 it, 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 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”, while 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 that they hold 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 Language opposed to Statically Typed (Compiled) Language. 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 too much of a hassle for developers of Dynamically 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), the inputs and output of a function 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 not 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 Enforce¹ to check for correctness of the code. The Python `typing` library offers a built-in arrange 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 and 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.

¹GitHub 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 become 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 of 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:

```
1 import numpy as np
2 A = np.array([
3     [1., 2., 4.],
4     [6., 3., 2.],
5     [6., 9., 5.],
6 ])
7
8 A /= np.max(A)
```

```
1 A = [
2     [1., 2., 4.],
3     [6., 3., 2.],
4     [6., 9., 5.],
5 ]
6
7 m = max(map(max, A))
8 for i in range(3):
9     for j in range(3):
10        A[i][j] /= m
```

Multidimensional arrays, arrays which are indexed not by one but by n 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 tensor 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 NumPy². For example, all of the following variables hold a valid NumPy array:

²Notice that this represents a challenge for Static Type Analysis as it requires 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.

```

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

```

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

```

Reshaping preserves the number of elements in an array. If a `-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:

```

1 a = np.zeros((10, 3, 4)) + 2
2 b = np.arange(3).reshape((3, 1))
3 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 they 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, 2005, 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)).

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 **overapproximation**

of some property of a program. To explain what an overapproximation is, consider the following piece of code:

```
1 a = int(input())
2 a = abs(a)
3 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 trade-off 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 warranted 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. (2005, Chapters 1 and 4), Cousot and Cousot (1977), Nipkow and 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.

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:

```

1 cond1 = bool(input()) # Stdin input
2 cond2 = bool(input())
3
4 if cond1:
5     a = 0
6     b = 20
7 elif cond2:
8     a = 2
9     b = 6
10 else:
11     a = 1
12     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.

It is clear that computing with sets of values is expensive, even impossible at times, e.g. the set $\{a \in \mathbb{N} \mid a \leq 0\}$ 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 trade-off 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:

```

1 b = bool(input())
2
3 if b:
4     a = 2
5     b = -1
6 else:
7     a = 6
8     b = 3

```



```

9 |
10 | c = a / b # c is either -2 or 2

```

It is easy for us to run this example in all possible inputs, as there are only two. 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 zero³.

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 overapproximation of all possible values⁴. 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 assignment `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 (opposed to maybe `True` comparisons)⁶.

The formal definition of an overapproximation is an Abstract Domain. An Abstract Domain is composed of a lattice, a concretisation function, an abstraction function, 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 \perp .

Given two lattices, $(L_1, \leq_{L_1}, \sqcap_{L_1}, \sqcup_{L_1}, \top_{L_1}, \perp_{L_1})$ and $(L_2, \leq_{L_2}, \sqcap_{L_2}, \sqcup_{L_2}, \top_{L_2}, \perp_{L_2})$, we can define α and γ , where:

- $\alpha : L_1 \rightarrow L_2$ called the abstraction function.
- $\gamma : L_2 \rightarrow L_1$ called the concretisation function.

L_1 is often called the Concrete Domain and L_2 is called the Abstract Domain. In this work, the Concrete Domain will always be defined as the power set of the values in the language with \in as the order operator.

³Dividing by zero throws an Exception in Python.

⁴Assuming no built-in value has been changed prior to the execution of the code.

⁵The sign overapproximation (or Sign Interval Abstract Domain) has only three possible 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 impossible for us to determine a unique value using Sign Interval Domain other than the case `0`.

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

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

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

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

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:

```

1  if _:
2    a = 2
3  else:
4    a = 4
5  a -= 6

```

We know that the value of `a` in the *then* branch 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` lies 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:

```

1  a = 4
2  while not condition(a):
3    a += 3

```

Unrolling the loop, we get:

```

1  a = 4      # a ∈ [4, 4]
2
3  a += 3    # a ∈ [4, 7]
4  a += 3    # a ∈ [4, 10]
5  a += 3    # a ∈ [4, 13]
6  a += 3    # a ∈ [4, 16]

```

```

7  ...
8  # a ∈ [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, and 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.

It is not possible to warranty to find a fixed point by just evaluating a piece of code over and over (the body of the while loop). Thus, we have the need of finding a mechanism that allows us to ensure that 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 before⁷:

$$[4, 4][4, 7][4, 10][4, 13][4, 16][4, 19][4, 22][4, 25] \dots$$

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 on an increasing sequence it will, after a finite amount of steps, find a fixed point (it 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 and over we will get:

$$[4, 7][4, 10][4, 13][4, 16][4, \infty][4, \infty][4, \infty] \dots$$

Notice that we found a fixed point, namely, $[4, \text{inf}]$. 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. This is the case for this work, no single Abstract Domain defined in here requires widening and narrowing operators, 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 let 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:

$$State : Var \rightarrow Val$$

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

$$State^\# : Var \rightarrow Val^\#$$

If the `State` of the program is defined as a function from *Vars* into their values ($State : Var \rightarrow Val$), and $Val^\#$ is the Abstract Domain for values in the languages, then the function $State^\# : Var \rightarrow Val^\#$ is an Abstract Domain⁸ for the state of the program.

As an example consider:

```

1  int main() {
2      int cond1, cond2, a, b, c;
3      std::cin » cond1 » cond2;
4
5      if (cond1) {
6          a = 0;
7          b = 20;
8      } else if (cond2) {
9          a = 2;
10         b = 6;
11     } else {
12         a = 1;
13         b = 9;
14     }
15     c = a + b;
16     return 0;
17 }
```

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:

```

1  // T: means Top
2  int main() {
3      int cond1, cond2, a, b, c;
4      // {'cond1': T, 'cond2': T, 'a': T, 'b': T, 'c': T}
5      std::cin » cond1 » cond2;
6      // {'cond1': T, 'cond2': T, 'a': T, 'b': T, 'c': T}
7
8      if (cond1) {
9          a = 0;
10         // {'cond1': T, 'cond2': T, 'a': 0, 'b': T, 'c': T}
11         b = 20;
12         // {'cond1': T, 'cond2': T, 'a': 0, 'b': 20, 'c': T}
13     } else {
14         if (cond2) {
```

⁸For a deeper look and a proof of this statement see Nielson et al. (2005), Subchapter 4.4.

```

15     a = 2;
16     // {'cond1': 0, 'cond2': T, 'a': 2, 'b': T, 'c': T}
17     b = 6;
18     // {'cond1': 0, 'cond2': T, 'a': 2, 'b': 6, 'c': T}
19 } else {
20     a = 1;
21     // {'cond1': 0, 'cond2': 0, 'a': 1, 'b': T, 'c': T}
22     b = 9;
23     // {'cond1': 0, 'cond2': 0, 'a': 1, 'b': 9, 'c': T}
24 }
25 // {'cond1': 0, 'cond2': T, 'a': [1,2], 'b': [6,9], 'c': T}
26 }
27 // {'cond1': T, 'cond2': T, 'a': [0,2], 'b': [6,20], 'c': T}
28 c = a + b;
29 // {'cond1': T, 'cond2': T, 'a': [0,2], 'b': [6,20], 'c': [6,22]}
30 return 0;
31 }

```

Excellent! We have done it! We have an 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

Abstract Semantics is the name that is given to the semantics that operate with abstract values (from the Abstract Domain), as opposed to Concrete Semantics that work with the original (concrete) values. To define these semantics the original (concrete) semantics and the Abstract values are required. The process of defining the abstract semantics involves applying the concretisation and abstraction functions onto the semantic rules we want to transform.

Often, this process is straightforward and no need to use the abstraction or concretisation functions is necessary, all that is needed is to understand how each operation from the Concrete Domain translates to the new Domain. For example, the abstract operation $+$ between two (interval) integer numbers is defined as:

$$[a, b] +^\# [c, d] := [a + c, b + d]$$

The semantics of interval arithmetics are well studied and can be easily implemented from theory (Hayes, 2003).

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):

```

1 // {'a': [2,4], 'b': T}
2 if (a < 3) {
3     // Applying backward assignment: a < 3

```

```
4 // {'a': 2, 'b':  $\top$ }
5 b = a;
6 // {'a': 2, 'b': 2}
7 } else {
8 // Applying backward assignment:  $a \geq 3$ 
9 // {'a': [3,4], 'b':  $\top$ }
10 b = a - 2;
11 // {'a': [3,4], 'b': [1,2]}
12 }
13 // {'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.

Another characteristic that will not be used in this work is relational Abstract Domains. A relational Abstract Domain keeps some of the relationships between variables which means that their overapproximations are tighter than those of non-relational overapproximations. We will only use non-relational Abstract Domains. For a detailed explanation on relational and non-relational Abstract Domains see Miné (2004).

Chapter 3

Abstract Interpretation for Python

In this chapter, an Abstract Interpreter for Python is proposed. The chapter is divided into five parts: definition of semantics, concrete semantics for Value Analysis of Python code, abstract semantics from the concrete semantics, an extension to abstract semantics to work with NumPy arrays, and the abstract semantics of type annotations. This chapter is intended as a guide on how the Abstract Interpreter was built. A deeper look at the gory details of the concrete and abstract semantics are given in Appendix A.

3.1 Semantics

In Mathematics, the word “semantics” refers to the meaning of sentences in formal languages (Gunter, 1992). The semantics of programming languages consist of what a piece of code means, how to interpret it, and how to analyse it.

Mitchell (1996) classifies programming languages semantics into three categories: axiomatic, denotational, and operational. Axiomatic semantics describe the behaviour of code with mathematical logic. The goal of denotational semantics is to find a mathematical “object” that represents what a program does. The program meaning is given by the object properties or lack thereof. 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.

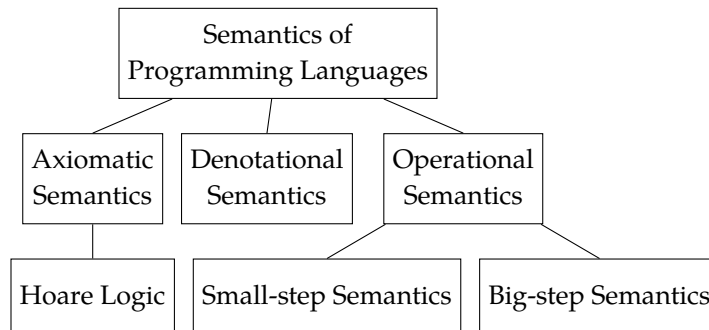


FIGURE 3.1: Classification of programming languages semantics by Mitchell (1996)

On the one hand, denotational semantics must not provide a way to determine an implementation. On the other hand, an implementation can be easily built from operational semantics. Small-step semantics, one kind of operational semantics, define what the program does by describing a set of rules over the syntax of the language. The meaning of the program is given by the application of the rules. The set of rules is often a specification of what a compiler does. Thus, it is not surprising that all efforts to define a formal semantics

for Python have been small-step semantics as it is easy to define rules from the translation process from code to “binary” (Ranson et al., 2008; Guth, 2013; Politz et al., 2013; Fromherz et al., 2018).

For this reason, we define in this work some small-steps semantics for Python. We define them from the execution model of CPython, that is, the official implementation of the Python language. In the following two sections, we define the small-step semantics of a portion of Python, our concrete semantics for the problem; and, based on them we will define some abstract semantics for Python.

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 large language, thus we will take just a portion of it into account. In Figure 3.2, we present the subset of Python that we intend to study in this work.

$expr$	$:=$	$i \in \mathbb{Z} \mid j \in \text{float} \mid \text{True} \mid \text{False} \mid \text{None}$	primitive values
		$ $ $identifier$	variable name
		$ $ $expr \ op \ expr \mid expr \ cmpop \ expr$	binary operations
		$ $ $expr(expr \ *)$	function call
		$ $ $expr.identifier$	attribute access
		$ $ $expr[expr]$	subscript access
		$ $ $[expr \ *]$	lists
		$ $ $(expr \ *)$	tuples
$stmt$	$:=$	$\text{del } expr$	delete
		$ $ $expr = expr$	assignment
		$ $ $expr \ op = \ expr$	augmented assignment
		$ $ $expr: expr = expr$	type annotation
		$ $ $\text{while } expr: stmt^+$	loop
		$ $ $\text{if } expr: stmt^+ \text{ else: } stmt^+$	if-else
		$ $ $\text{import } alias^+ \mid \text{from } identifier \text{ import } alias^+$	import library
		$ $ $expr$	
op	$:=$	$+ \mid - \mid * \mid / \mid \% \mid ** \mid \ll \mid \gg \mid //$	
$cmpop$	$:=$	$< \mid <= \mid > \mid >=$	
$alias$	$:=$	$identifier \mid identifier \text{ as } identifier$	

FIGURE 3.2: Reduced Python syntax used throughout this work.

We intentionally omit any details related to the spacing of blocks in Python, as our main focus is the meaning of Python values and not how blocks of statements are defined in the language.

We follow the same approach taken by (Fromherz et al., 2018) and define the state of the program with a global scope and a heap. Python memory is represented as a function from memory positions (**Addr**) to python values (**Val**). The global scope is defined as a function from identifiers (**Iden**) to memory positions (**Addr**).

$$\begin{aligned} \text{Global} &:= \text{Iden} \rightarrow \text{Addr} \cup \{\text{Undefined}\} \\ \text{Heap} &:= \text{Addr} \rightarrow \text{Val} \cup \{\text{Undefined}\} \end{aligned}$$

In Python, everything is an object: numbers, lists, classes and even functions. We separate python values into three categories: primitive values, like numbers, booleans and `None`; Objects, like lists and tuples; and, primitive functions, like `int.__add__`.

$$\begin{aligned}
\mathbf{Val} &:= \mathbf{PrimVal} \mid \mathbf{Object} \mid \langle \mathbf{prim-callable} \rangle \\
&\quad \mid \mathbf{Addr} \times \langle \mathbf{prim-callable} \rangle \quad \text{function associated to value} \\
\mathbf{PrimVal} &:= i \in \mathbb{Z} \mid j \in \text{float} \mid \text{True} \mid \text{False} \mid \text{None} \\
\mathbf{Object} &:= \mathbf{Type} \times \mathbf{Addr} \times (\mathbf{Key} \rightarrow \mathbf{Addr} \cup \{\text{Undefined}\}) \\
\mathbf{Type} &:= \text{List} \mid \text{Tuple} \\
\mathbf{Key} &:= \mathbf{Iden} + (\text{string} \times \mathbf{Val})
\end{aligned}$$

We can now define the rules that describe the behaviour of a program, that is, the small-step semantics of the language. The equations $\mathbb{E}[\![expr]\!]$ and $\mathbb{S}[\![stmt]\!]$ from Figure 3.3 define the small-step semantics.

$ \begin{aligned} \mathbb{E}[\![expr]\!] &: \mathbf{Global} \times \mathbf{Heap} \\ &\rightarrow \mathbf{Global} \times \mathbf{Heap} \times \mathbf{Val} \\ \mathbb{E}[\![identifier]\!] (\mathcal{G}, \mathcal{H}) &:= \\ &\text{if } \mathcal{G}(identifier) = \text{Undefined} \\ &\text{then } \langle \mathbf{Execution Halt} \rangle \\ &\text{else } (\mathcal{G}, \mathcal{H}, \mathcal{H}(\mathcal{G}(identifier))) \\ \mathbb{E}[\![n]\!] (\mathcal{G}, \mathcal{H}) &:= (\mathcal{G}, \mathcal{H}, n) \quad \text{for all } n \in \mathbb{Z} \\ \mathbb{E}[\![expr_1 \text{ op } expr_2]\!] (\mathcal{G}, \mathcal{H}) &:= \\ &\text{let } (\mathcal{G}_1, \mathcal{H}_1, v_1) := \mathbb{E}[\![expr_1]\!] (\mathcal{G}, \mathcal{H}) \\ &\quad (\mathcal{G}_2, \mathcal{H}_2, v_2) := \mathbb{E}[\![expr_2]\!] (\mathcal{G}_1, \mathcal{H}_1) \\ &\quad \text{prim_op} := \text{get_prim_op}(op, v_1, v_2) \\ &\text{in } \text{prim_op}(\mathcal{G}_2, \mathcal{H}_2) \end{aligned} $	$ \begin{aligned} \mathbb{S}[\![stmt]\!] &: \mathbf{Global} \times \mathbf{Heap} \rightarrow \mathbf{Global} \times \mathbf{Heap} \\ \mathbb{S}[\![stmt_1; stmt_2]\!] (\mathcal{G}, \mathcal{H}) &:= \\ &\text{let } (\mathcal{G}_1, \mathcal{H}_1) := \mathbb{S}[\![stmt_1]\!] (\mathcal{G}, \mathcal{H}) \\ &\text{in } \mathbb{S}[\![stmt_2]\!] (\mathcal{G}_1, \mathcal{H}_1) \\ \mathbb{S}[\![id = expr]\!] (\mathcal{G}, \mathcal{H}) &:= \\ &\text{let } (\mathcal{G}_1, \mathcal{H}_1, val) := \mathbb{E}[\![expr]\!] (\mathcal{G}, \mathcal{H}) \\ &\quad \text{adnew} := \text{get_free_addr}(\mathcal{H}_1) \\ &\text{in } (\mathcal{G}_1[id \rightarrow \text{adnew}], \mathcal{H}_1[\text{adnew} \rightarrow val]) \\ \mathcal{G}_{empty} &: \mathbf{Global} \\ \mathcal{G}_{empty} &:= \text{Undefined} \\ \mathcal{H}_{empty} &: \mathbf{Heap} \\ \mathcal{H}_{empty} &:= \text{Undefined} \end{aligned} $
---	---

FIGURE 3.3: A portion of the Small-steps semantics for Python. The functions *get_prim_op* and *get_free_addr* are defined in Appendix A

The construction $\langle \mathbf{Execution Halt} \rangle$ indicates that the evaluation cannot go any further, i.e. the interpreter will stop and throw an exception. For example, accessing to a variable that has not yet been defined is considered an error (Python will throw an exception if this happens).

The meaning of our program is given by the function $\mathbb{S}[\![stmt]\!]$ with type $\mathbf{State} \rightarrow \mathbf{State}$ (where $\mathbf{State} := \mathbf{Global} \times \mathbf{Heap}$). In this work, we are interested on the values that result from the application of the function $\mathbb{S}[\![stmt]\!]$ on a clean state, which is a state that has only built-in variables defined. Statements modify the state of the program while expressions modify the state of the program and return a python value as result, e.g. the expression `ls.pop()` removes the last element in the list `ls` and returns the value contained. This is the reason why the type of the function $\mathbb{E}[\![expr]\!]$ is $\mathbf{State} \rightarrow \mathbf{State} \times \mathbf{Val}$.

Let us consider the simple program:

```

1  b = 2
2  a = b + 3

```

We can apply the semantics rules on the example by evaluating $\mathbb{S}[\![b = 2; a = b + 3]\!] (\mathcal{G}_{empty}, \mathcal{H}_{empty})$. Figure 3.4 presents the step by step of the evaluation.

$$\begin{aligned}
& \mathbb{S} \llbracket b = 2; a = b + 3 \rrbracket (\mathcal{G}_{empty}, \mathcal{H}_{empty}) \\
& \stackrel{\text{def}}{=} \text{let } (\mathcal{G}_1, \mathcal{H}_1) := \mathbb{S} \llbracket b = 2 \rrbracket (\mathcal{G}_{empty}, \mathcal{H}_{empty}) \\
& \quad \text{in } \mathbb{S} \llbracket a = b + 3 \rrbracket (\mathcal{G}_1, \mathcal{H}_1) \\
& \left| \begin{aligned}
& \mathbb{S} \llbracket b = 2 \rrbracket (\mathcal{G}_{empty}, \mathcal{H}_{empty}) \\
& \stackrel{\text{def}}{=} \text{let } (\mathcal{G}_1, \mathcal{H}_1, val) := \mathbb{E} \llbracket 2 \rrbracket (\mathcal{G}_{empty}, \mathcal{H}_{empty}) \\
& \quad adnew := get_free_addr(\mathcal{H}_1) \\
& \quad \text{in } (\mathcal{G}_1[b \mapsto adnew], \mathcal{H}_1[adnew \mapsto val]) \\
& \stackrel{\text{def}}{=} \text{let } (\mathcal{G}_1, \mathcal{H}_1, val) := (\mathcal{G}_{empty}, \mathcal{H}_{empty}, 2) \\
& \quad adnew := get_free_addr(\mathcal{H}_1) \\
& \quad \text{in } (\mathcal{G}_1[b \mapsto adnew], \mathcal{H}_1[adnew \mapsto val]) \\
& = \text{let } adnew := get_free_addr(\mathcal{H}_{empty}) \\
& \quad \text{in } (\mathcal{G}_{empty}[b \mapsto adnew], \mathcal{H}_{empty}[adnew \mapsto 2]) \\
& = \text{let } adnew := 0 \\
& \quad \text{in } (\mathcal{G}_{empty}[b \mapsto adnew], \mathcal{H}_{empty}[adnew \mapsto 2]) \\
& = (\mathcal{G}_{empty}[b \mapsto 0], \mathcal{H}_{empty}[0 \mapsto 2]) \\
& = ([b \mapsto 0], [0 \mapsto 2]) \quad \text{simplifying notation}
\end{aligned} \right. \\
& = \text{let } (\mathcal{G}_1, \mathcal{H}_1) := ([b \mapsto 0], [0 \mapsto 2]) \\
& \quad \text{in } \mathbb{S} \llbracket a = b + 3 \rrbracket (\mathcal{G}_1, \mathcal{H}_1) \\
& = \mathbb{S} \llbracket a = b + 3 \rrbracket ([b \mapsto 0], [0 \mapsto 2]) \\
& \stackrel{\text{def}}{=} \text{let } (\mathcal{G}_1, \mathcal{H}_1, val) := \mathbb{E} \llbracket b + 3 \rrbracket ([b \mapsto 0], [0 \mapsto 2]) \\
& \quad adnew := get_free_addr(\mathcal{H}_1) \\
& \quad \text{in } (\mathcal{G}_1[a \mapsto adnew], \mathcal{H}_1[adnew \mapsto val]) \\
& = \dots = \\
& = \text{let } (\mathcal{G}_1, \mathcal{H}_1, val) := ([b \mapsto 0], [0 \mapsto 2], 5) \\
& \quad adnew := get_free_addr(\mathcal{H}_1) \\
& \quad \text{in } (\mathcal{G}_1[a \mapsto adnew], \mathcal{H}_1[adnew \mapsto val]) \\
& = \text{let } adnew := get_free_addr([0 \mapsto 2]) \\
& \quad \text{in } ([b \mapsto 0][a \mapsto adnew], [0 \mapsto 2][adnew \mapsto 5]) \\
& = ([b \mapsto 0][a \mapsto 1], [0 \mapsto 2][1 \mapsto 5]) \\
& = ([b \mapsto 0, a \mapsto 1], [0 \mapsto 2, 1 \mapsto 5])
\end{aligned}$$

FIGURE 3.4: Evaluation of the program `b = 2; a = b + 3` with the small-step semantics from Figure 3.3

At the end of the execution, the state of the program is:

$\mathcal{G}_{output} : \mathbf{Iden} \rightarrow \mathbf{Addr} \cup \{\text{Undefined}\}$	$\mathcal{H}_{output} : \mathbf{Addr} \rightarrow \mathbf{Val} \cup \{\text{Undefined}\}$
'a' \mapsto 1	0 \mapsto 2
'b' \mapsto 0	1 \mapsto 5
id \mapsto Undefined	n \mapsto Undefined

This is telling us that the variables `a` and `b` hold the values `2` and `5`, respectively.

Notice that we have not yet defined the semantics for the general case $\mathbb{S}[\![expr_1 = expr_2]\!]$ but only for the case $\mathbb{S}[\![identifier = expr]\!]$. Defining the small-step semantics on the syntax presented before is inconvenient. To see why working with this syntax is inconvenient, observe that the delete statement, `del expr`, does not expect a value from the evaluation of the expression `expr` but a name or position in memory. Both `del a` and `ls = [1,9,4,5]; del ls[2]` are valid python statements, but their semantics depend on the expression being evaluated. Inversely, the return value of evaluating expressions depends on the context they are being called, the expression `ls[2]` returns a value or a position in memory depending on the context is being used. `4 + ls[2]`, `ls[2] = 4` and `del ls[2]` are examples of the three different contexts where `ls[2]` can be called.

How could an expression like `ls[2]` know the context where it is going to be used? We opted to follow what CPython does. CPython extends the syntax of expressions to allow them to know the context from where they are being called. The new (extended) syntax can be seen in Figure 3.5.

$expr$	$:=$	Int(i) for $i \in \mathbb{Z}$ Float(j) for $j \in \text{float}$	
		True False None	primitive values
		Name($identifier, expr_context$)	variable name
		BinOp($op, expr, expr$)	
		Compare($expr, cmpop, expr$)	binary operations
		Call($expr, expr^*$)	function call
		Attribute($expr, identifier$)	attribute access
		Subscript($expr, expr, expr_context$)	subscript access
		List($expr^*$)	lists
		Tuple($expr^*$)	tuples
$stmt$	$:=$	Delete($expr$)	delete
		Assign($expr, expr$)	assignment
		AugAssign($expr, op, expr$)	augmented assignment
		AnnAssign($expr, expr, expr$)	type annotation
		While($expr, stmt^+$)	loop
		If($expr, stmt^+, stmt^+$)	if
		Import($alias^+$)	import library
		ImportFrom($identifier, alias^+$)	import from
		Expr($expr$)	
$expr_context$	$:=$	Load Store Del	
op	$:=$	Add Sub Mult Div Mod Pow	
		LShift RShift FloorDiv	
$cmpop$	$:=$	Lt LtE Gt GtE	
$alias$	$:=$	identifier (identifier, identifier)	

FIGURE 3.5: Python syntax used to define the semantics of the Abstract Interpreter.

This is the same syntax that is used by the parser implemented by CPython (Van Rossum et al., 2007). The developer is not supposed to write in this new syntax. The parser takes code written in the first syntax and defines a proper context (Load, Store, or Del) for the expressions `Name` and `Subscript`. All other expressions are considered to have context Load by default.

The following is the informal meaning of each context:

Load indicates that the result of evaluating the expression will be used in further computations. For example, the result of evaluating the expression `b` in `b + 2` is a value.

Store indicates that we want to save a value into the expression. For example, the result of evaluating the expression `b` in `b = 2` is the location of `b` in the global scope.

Del indicates that we want to delete the expression from where it is stored. For example, the result of evaluating the expression `b` in `del b` is the name that identifies the variable, `b`.

In CPython, the expressions `Attribute`, `Subscript` and `Name` have a context as part of their definition. In our specification, only `Subscript` and `Name` have a context as we are not interested in modifying or deleting the value of any attribute. No attribute for built-in functions and objects can be modified or deleted. For example, given the list `a = [2, 3]`, the statements `del a.insert` and `a.insert = 3` are erroneous in Python, thus we have no need to allow `Attribute`s to be deleted or modified.

In Figure 3.6, the small-step semantics for the new syntax are presented.

Notice the increase in the number of `<Execution Halt>`s in the semantics. These increase is due to all possible errors a parser may make. The Abstract Interpreter will also fail if there is a failure in the parser, we suppose that the parser works as intended.

Expressions return one of three different values **Val**, **Object** \times (**string** \times **Val**) or **Iden**. The value returned depends on the context the expression is called and how is it accessed:

- **Val**: is returned when the expressions `Subscript` and `Name` have context Load. All other expressions return only Vals.
- **Iden**: is returned when the expression `Name` has context Del or Store.
- **Object** \times (**string** \times **Val**): is returned when the expression `Subscript` has context Del or Store.

3.3 Abstract Semantics

In this section, we will define the abstract semantics of the language. To define them, we need to define first an Abstract Domain for the state of the program. Recall that Concrete semantics work on Python values (numbers, constants, functions, lists, etc) and abstract semantics work on Abstract Python values.

First, we will define the Abstract Python values. At last, we will show the new semantics, the abstract semantics.

3.3.1 State Abstract Domain

Before we define the Abstract Domain for a state of the program, **Global** \times **Heap**, we will define the Abstract Domain for a primitive value in the language. Concrete primitive values are defined as

$$\mathbf{PrimVal}^\# := \text{Int}^\# \mid \text{Float}^\# \mid \text{Bool}^\# \mid \text{None}^\# \mid \top_{\text{PrimVal}} \mid \perp_{\text{PrimVal}}$$

The Abstract Domain for primitive values is constructed from individual Abstract Domains, one Abstract Domain per each one of the types of values that **PrimVal** holds as it can be seen in Figure 3.7.

$$\begin{aligned}
\mathbb{E}[\![expr]\!] &: \mathbf{Global} \times \mathbf{Heap} \\
&\rightarrow \mathbf{Global} \\
&\times \mathbf{Heap} \\
&\times \mathbf{Val} \cup (\mathbf{Object} \times (\mathbf{string} \times \mathbf{Val})) \cup \mathbf{Iden} \\
\mathbb{E}[\![\mathbf{Name}(id)]\!] (\mathcal{G}, \mathcal{H}) &:= \\
\text{match } ctx \text{ in} & \\
\text{case Load} &\rightarrow \text{if } \mathcal{G}(id) = \mathbf{Undefined} \\
&\quad \text{then if } isbuiltin(id) \\
&\quad \quad \text{then } (\mathcal{G}, \mathcal{H}, \langle \mathbf{builtin-val} \rangle(id)) \\
&\quad \quad \text{else } \langle \mathbf{Execution Halt} \rangle \\
&\quad \text{else } (\mathcal{G}, \mathcal{H}, \mathcal{H}(\mathcal{G}(id))) \\
\text{case Store} &\rightarrow (\mathcal{G}, \mathcal{H}, id) \quad \text{to be assigned} \\
\text{case Del} &\rightarrow (\mathcal{G}, \mathcal{H}, id) \quad \text{to be deleted} \\
\mathbb{E}[\![\mathbf{Int}(n)]\!] (\mathcal{G}, \mathcal{H}) &:= (\mathcal{G}, \mathcal{H}, n) \\
\mathbb{E}[\![\mathbf{BinOp}(op, a, b)]\!] (\mathcal{G}, \mathcal{H}) &:= \\
\text{let } (\mathcal{G}_1, \mathcal{H}_1, v_1) &:= \mathbb{E}[\![a]\!] (\mathcal{G}, \mathcal{H}) \\
(\mathcal{G}_2, \mathcal{H}_2, v_2) &:= \mathbb{E}[\![b]\!] (\mathcal{G}_1, \mathcal{H}_1) \\
prim_op &:= get_prim_op(op, v_1, v_2) \\
\text{in if } kind(v_1) \neq \mathbf{Val} \vee kind(v_2) \neq \mathbf{Val} & \\
\text{then } \langle \mathbf{Execution Halt} \rangle & \\
\text{else let } prim_op &:= get_prim_op(op, v_1, v_2) \\
\text{in } prim_op(\mathcal{G}_2, \mathcal{H}_2) & \\
\mathbb{S}[\![\mathbf{Assign}(var, val)]\!] (\mathcal{G}, \mathcal{H}) &:= \\
\text{let } (\mathcal{G}_1, \mathcal{H}_1, ass) &:= \mathbb{E}[\![var]\!] (\mathcal{G}, \mathcal{H}) \\
(\mathcal{G}_2, \mathcal{H}_2, rightval) &:= \mathbb{E}[\![val]\!] (\mathcal{G}_1, \mathcal{H}_1) \\
rval &:= \text{if } is_value(rightval) \text{ then } val \text{ else } \langle \mathbf{Execution Halt} \rangle \\
\text{in match } ass \text{ in} & \\
\text{case } id : \mathbf{Val} &\rightarrow (\mathcal{G}_2[ass \mapsto rval], \mathcal{H}_2) \\
\text{case } ((t, addr, o) : \mathbf{Object}, ('index', val : \mathbf{Val})) &\rightarrow \\
\text{let } setindex &:= get_prim_set_index(t) \\
\text{in } setindex(\mathcal{G}_2, \mathcal{H}_2, o, val, addr, rval) & \\
\text{case otherwise} &\rightarrow \langle \mathbf{Execution Halt} \rangle
\end{aligned}$$

FIGURE 3.6: Small-step (concrete) semantics for Python using syntax from Figure 3.5.

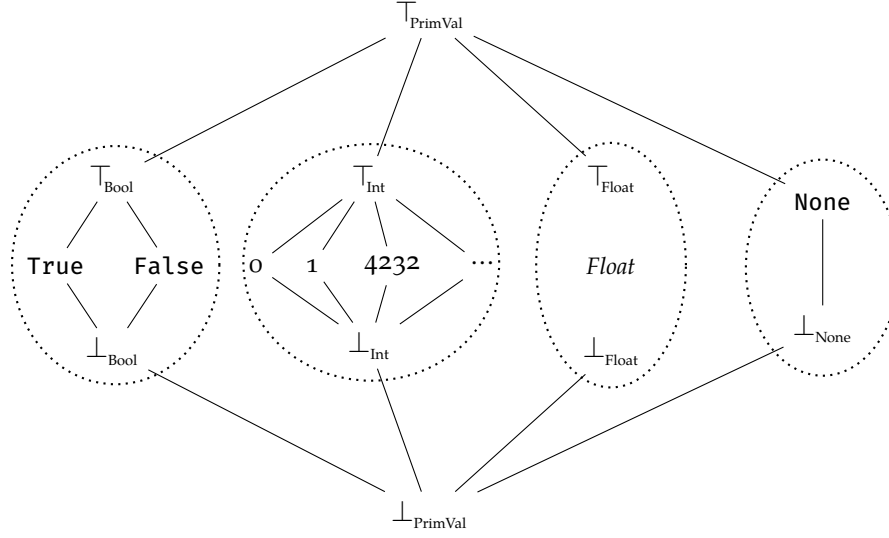


FIGURE 3.7: Lattice/Abstract Domain for primitive values.

In this Abstract Domain, joining any two differing values gives us a \top value. If both values are of the same “type”, e.g. both are `Int`s, then the \top will be of that type. For example, $5 \cup 6 = \top_{\text{Int}}$ and $5 \cup 6.2 = \top_{\text{PrimVal}}$.

The selected Abstract Domains for integers and floating point numbers are the simplest Abstract Domains for numbers that work for Value Analysis. We do not use more complex Abstract Domains, like intervals, because there is no need for them to analyse the shape of tensors.

Recall that the semantics of a program work on states, thus we need to define an abstract value for the abstract semantics to work with. Remember that the state of a program is a value of type **Global** \times **Heap**. The abstract state of a program is defined as **Global**[#] \times **Heap**[#], where:

$$\begin{aligned}
 \text{Val}^\# &::= \text{PrimVal}^\# \mid \text{Object}^\# \mid \langle \text{prim-callable} \rangle^\# \\
 \text{PrimVal}^\# &::= \text{Int}^\# \mid \text{Float}^\# \mid \text{Bool}^\# \mid \text{None}^\# \mid \top_{\text{PrimVal}} \mid \perp_{\text{PrimVal}} \\
 \text{Object}^\# &::= \text{Type} \times \text{Addr} \times (\text{Key} \rightarrow \text{Addr} \cup \{\text{Undefined}\}) \\
 \text{Type} &::= \text{List} \mid \text{Tuple} \\
 \text{Key} &::= \text{Iden} + (\text{string} \times \text{Val}^\#)
 \end{aligned}$$

Explaining in detail how the Abstract Domain for the state is defined is out of the scope of this chapter. A throughout explanation on the construction of the Abstract Domain can be seen in Appendix A. We will dedicate the rest of this section to motivate why this representation has been chosen.

Consider the following piece of code:

```

1  if s:
2      a = 2
3      b = a + 3
4      c = [0.0, a, b]
5  else:
6      a = 3
7      b = a + 2
8      c = [1, a, b]

```

In the example, we do not know the value of `s`, and therefore, we have no idea which of the two branches of the `if` statement will be executed. The solution that Abstract Interpretation presents us is to run both branches separately, each with a copy of the state of the program. Then, all we need to do is to find an **overapproximation** of the two new states. Fortunately, finding an overapproximation of the states is what we built an Abstract Domain for. The $join (\cup^{State})$ operation of an Abstract Domain tells us the smallest common overapproximation between the two states.

We know that the state at the end of the “then” branch is $(\mathcal{G}_1, \mathcal{H}_1)$, and the state at the end of the “else” branch is $(\mathcal{G}_2, \mathcal{H}_2)$, where:

$$\begin{aligned} \mathcal{G}_1(i) &= \begin{cases} 0 & \text{if } i = 's' \\ 1 & \text{if } i = 'a' \\ 2 & \text{if } i = 'b' \\ 3 & \text{if } i = 'c' \\ \text{Undefined} & \text{otherwise} \end{cases} & \mathcal{G}_2(i) &= \begin{cases} 0 & \text{if } i = 's' \\ 1 & \text{if } i = 'a' \\ 2 & \text{if } i = 'b' \\ 3 & \text{if } i = 'c' \\ \text{Undefined} & \text{otherwise} \end{cases} \\ \mathcal{H}_1(n) &= \begin{cases} \top_{\text{Val}} & \text{if } n = 0 \\ 2 & \text{if } n = 1 \\ 5 & \text{if } n = 2 \\ (\text{List}, 3, l_1) & \text{if } n = 3 \\ 3 & \text{if } n = 4 \\ 0.0 & \text{if } n = 5 \\ 2 & \text{if } n = 6 \\ 5 & \text{if } n = 7 \\ \text{Undefined} & \text{otherwise} \end{cases} & \mathcal{H}_2(n) &= \begin{cases} \top_{\text{Val}} & \text{if } n = 0 \\ 3 & \text{if } n = 1 \\ 5 & \text{if } n = 2 \\ (\text{List}, 3, l_2) & \text{if } n = 3 \\ 3 & \text{if } n = 4 \\ 1 & \text{if } n = 5 \\ 3 & \text{if } n = 6 \\ 5 & \text{if } n = 7 \\ \text{Undefined} & \text{otherwise} \end{cases} \\ l_1(i) &= \begin{cases} 4 & \text{if } i = 'size' \\ 5 & \text{if } i = ('index', 0) \\ 6 & \text{if } i = ('index', 1) \\ 7 & \text{if } i = ('index', 2) \\ \text{Undefined} & \text{otherwise} \end{cases} & l_2(i) &= \begin{cases} 4 & \text{if } i = 'size' \\ 5 & \text{if } i = ('index', 0) \\ 6 & \text{if } i = ('index', 1) \\ 7 & \text{if } i = ('index', 2) \\ \text{Undefined} & \text{otherwise} \end{cases} \end{aligned}$$

The result of joining the two states is $(\mathcal{G}_{new}, \mathcal{H}_{new})$, where:

$$\begin{aligned} \mathcal{G}_{new}(i) &= \begin{cases} 0 & \text{if } i = 's' \\ 1 & \text{if } i = 'a' \\ 2 & \text{if } i = 'b' \\ 3 & \text{if } i = 'c' \\ \text{Undefined} & \text{otherwise} \end{cases} & \mathcal{H}_{new}(n) &= \begin{cases} \top_{\text{Val}} & \text{if } n = 0 \\ \top_{\text{Int}} & \text{if } n = 1 \\ 5 & \text{if } n = 2 \\ (\text{List}, 3, l_{new}) & \text{if } n = 3 \\ 3 & \text{if } n = 4 \\ \top_{\text{PrimVal}} & \text{if } n = 5 \\ \top_{\text{Int}} & \text{if } n = 6 \\ 5 & \text{if } n = 7 \\ \text{Undefined} & \text{otherwise} \end{cases} \\ l_{new}(i) &= \begin{cases} 4 & \text{if } i = 'size' \\ 5 & \text{if } i = ('index', 0) \\ 6 & \text{if } i = ('index', 1) \\ 7 & \text{if } i = ('index', 2) \\ \text{Undefined} & \text{otherwise} \end{cases} \end{aligned}$$

Now that we have defined our abstract state, we can define the abstract semantics.

3.3.2 Semantics for the Abstract State

Once we have defined the concrete semantics and Abstract State, defining abstract semantics is straightforward. All we need to do is to modify the concrete semantics to work with new values, see Figure 3.8.

$$\begin{aligned}
 \mathbb{E}^\sharp[\![expr]\!] &: \mathbf{Global}^\sharp \times \mathbf{Heap}^\sharp \\
 &\quad \rightarrow \mathbf{Global}^\sharp \\
 &\quad \times \mathbf{Heap}^\sharp \\
 &\quad \times (\mathbf{Val}^\sharp \cup (\mathbf{Object}^\sharp \times (\text{string} \times \mathbf{Val}^\sharp)) \cup \mathbf{Iden}) \\
 \\
 \mathbb{S}^\sharp[\![expr]\!] &: \mathbf{Global}^\sharp \times \mathbf{Heap}^\sharp \rightarrow \mathbf{Global}^\sharp \times \mathbf{Heap}^\sharp \\
 \\
 \mathbb{E}^\sharp[\![\mathbf{Name}(id)]\!] (\mathcal{G}^\sharp, \mathcal{H}^\sharp) &:= \\
 \text{match } ctx \text{ in} & \\
 \quad \text{case Load} &\rightarrow \text{if } \mathcal{G}^\sharp(id) = \text{Undefined} \\
 &\quad \text{then if } isbuiltin(id) \\
 &\quad \quad \text{then } (\mathcal{G}^\sharp, \mathcal{H}^\sharp, \langle builtin-val \rangle(id)) \\
 &\quad \quad \text{else let } ad := freeaddr(\mathcal{H}^\sharp) \\
 &\quad \quad \quad \text{in } (\mathcal{G}^\sharp[id \mapsto ad], \mathcal{H}^\sharp[ad \mapsto \top_{Val}], \top_{Val}) \\
 &\quad \quad \text{else } (\mathcal{G}^\sharp, \mathcal{H}^\sharp, \mathcal{H}^\sharp(\mathcal{G}^\sharp(id))) \\
 \quad \text{case Store} &\rightarrow (\mathcal{G}^\sharp, \mathcal{H}^\sharp, id) \quad \text{to be assigned} \\
 \quad \text{case Del} &\rightarrow (\mathcal{G}^\sharp, \mathcal{H}^\sharp, id) \quad \text{to be deleted} \\
 \\
 \mathbb{E}^\sharp[\![\mathbf{Int}(n)]\!] (\mathcal{G}^\sharp, \mathcal{H}^\sharp) &:= (\mathcal{G}^\sharp, \mathcal{H}^\sharp, n) \\
 \\
 \mathbb{S}^\sharp[\![\mathbf{Assign}(var, val)]\!] (\mathcal{G}, \mathcal{H}) &:= \\
 \text{let } (\mathcal{G}_1^\sharp, \mathcal{H}_1^\sharp, ass) &:= \mathbb{E}^\sharp[\![var]\!] (\mathcal{G}^\sharp, \mathcal{H}^\sharp) \\
 (\mathcal{G}_2^\sharp, \mathcal{H}_2^\sharp, rval) &:= \mathbb{E}^\sharp[\![val]\!] (\mathcal{G}_1^\sharp, \mathcal{H}_1^\sharp) \\
 &\quad rval := \text{if } is_value^\sharp(rval) \text{ then } val \text{ else } \langle \text{Execution Halt} \rangle \\
 \text{in match } ass \text{ in} & \\
 \quad \text{case } _ : \mathbf{Val} &\rightarrow (\mathcal{G}_2^\sharp[ass \mapsto rval], \mathcal{H}_2^\sharp) \\
 \quad \text{case } ((t, addr, o) : \mathbf{Object}, ('index', val : \mathbf{Val}^\sharp)) &\rightarrow \\
 \quad \quad \text{let } setindex^\sharp &:= get_prim_set_index^\sharp(t) \\
 \quad \quad \text{in } setindex^\sharp &(\mathcal{G}_2^\sharp, \mathcal{H}_2^\sharp, o, val, addr, rval) \\
 \quad \text{case otherwise} &\rightarrow \langle \text{Execution Halt} \rangle
 \end{aligned}$$

FIGURE 3.8: Excerpt of small-step abstract semantics for Python.

In the process of defining the new semantics, we can notice that the number of `<Execution Halt>` expressions have reduced. Every `<Execution Halt>` that is not the result of a parser error is caused by a type mismatch between values being operated. An example of an execution halt caused by a type mismatch is `5 + None` because no integer can be added to a `None`.

Executing a faulty piece of code now does not halt the interpreter:

```

1 | b = 2 + d
2 | if b < 3:
3 |     a = 2+3

```



```

4 else:
5   a = 5-1+1

```

The state result of evaluating the function $\mathbb{S}^\# \llbracket b = 2 + d; a = b < 3 \rrbracket$ is:

$$\mathcal{G}^\#(i) = \begin{cases} 0 & \text{if } i = 'd' \\ 1 & \text{if } i = 'b' \\ 2 & \text{if } i = 'a' \\ \text{Undefined} & \text{otherwise} \end{cases} \quad \mathcal{H}^\#(n) = \begin{cases} \top_{\text{Val}} & \text{if } n = 0 \\ \top_{\text{Val}} & \text{if } n = 1 \\ 5 & \text{if } n = 2 \\ \text{Undefined} & \text{otherwise} \end{cases}$$

3.4 NdArrays

The purpose of the specification 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 section, I extend the concrete semantics with NumPy arrays.

The following are all the things to take into account when extending our specification to handle a new type of “built-in” **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:

$(\text{NdArray}, n, s)$

where $n \in \mathbb{Z}$, and

$$s(i) = \begin{cases} n + 1 & \text{if } i = 'shape' \\ n + 5 + m + 3o & \text{if } i = ('index', m, o) \quad \forall m \in [0, 3], o \in [0, 2] \\ \text{Undefined} & \text{otherwise} \end{cases}$$

$$t(i) = \begin{cases} n + 2 & \text{if } i = 'size' \\ n + 3 & \text{if } i = ('index', 0) \\ n + 4 & \text{if } i = ('index', 1) \\ \text{Undefined} & \text{otherwise} \end{cases}$$

$$\mathcal{H}(m) = \begin{cases} (\text{NdArray}, n, s) & \text{if } m = n \\ (\text{Tuple}, n + 1, t) & \text{if } m = n + 1 \\ 2 & \text{if } m = n + 2 \\ 4 & \text{if } m = n + 3 \\ 3 & \text{if } m = n + 4 \\ 0 & \text{if } m = n + 5 + o \quad \forall o \in [0, 11] \\ \text{Undefined} & \text{otherwise} \end{cases}$$

Note: Remember that an **Object**[#] is a triple of the form **Type** × **Addr** × (**Key** → **Addr** ∪ {Undefined}).

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`.

$\langle \text{prim-callable} \rangle = \dots (\text{old operations}) \mid \langle \text{prim-np-zeros} \rangle \mid \langle \text{prim-np-dot} \rangle \mid \langle \text{prim-np-abs} \rangle \mid \dots$

Note: The values stored inside a NumPy array are considered 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 specification of the NumPy library primitives.

Nonetheless, defining formally each one of the NumPy functions above is fairly straightforward. Although, the hardest part of a formal definition 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 object¹.

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:

$$\begin{aligned} \langle \text{prim-np-size} \rangle (\mathcal{G}^\sharp, \mathcal{H}^\sharp) &:= \\ \text{let } (\mathcal{G}_1^\sharp, \mathcal{H}_1^\sharp, (\text{NdArray}, \text{addr}, \text{arr})) &:= \langle \text{prim-array} \rangle (\text{val}) (\mathcal{G}^\sharp, \mathcal{H}^\sharp) \\ (\text{Tuple}, \text{addr}, \text{tup}, \text{tup}) &:= \text{arr}('shape') \\ \text{in } \text{tup}('size') \end{aligned}$$

3. The NumPy module holding all operations is defined:

$$\langle \text{numpy-mod} \rangle := (\text{Module}, -1, f_{\text{numpy}})$$

where -1 indicates that the address is to be set only when the module is imported, and

$$f_{\text{numpy}}(s) = \begin{cases} \langle \text{prim-array} \rangle & \text{if } s = ('attr', 'array') \\ \langle \text{prim-dot} \rangle & \text{if } s = ('attr', 'dot') \\ \langle \text{prim-zeros} \rangle & \text{if } s = ('attr', 'zeros') \\ \langle \text{prim-ones} \rangle & \text{if } s = ('attr', 'ones') \\ \dots & \\ \text{Undefined} & \text{otherwise} \end{cases}$$

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

¹The NumPy function `array` takes almost anything as an input. `array` tries to interpret its input as an array in any way it can. There is no formal definition of how the values are interpreted although its semantics can be extracted by looking at its C implementation: <https://stackoverflow.com/a/40380014>

$$\begin{aligned}
\mathbb{S}^\#[\text{Import}(name)](\mathcal{G}^\#, \mathcal{H}^\#) &:= \\
&\text{match } name \text{ in} \\
&\text{case } ("numpy",) \rightarrow \\
&\quad \text{let } (\text{Module}, arr, mod) := \langle numpy - mod \rangle \\
&\quad \quad freeaddr := get_free_addr(\mathcal{H}^\#) \\
&\quad \text{in } (\mathcal{G}^\#[\text{'numpy'} \mapsto freeaddr], \mathcal{H}^\#[freeaddr \mapsto (\text{Module}, freeaddr, mod)]) \\
&\text{case } ("numpy", alias) \rightarrow \\
&\quad \text{let } (\text{Module}, arr, mod) := \langle numpy - mod \rangle \\
&\quad \quad freeaddr := get_free_addr(\mathcal{H}^\#) \\
&\quad \text{in } (\mathcal{G}^\#[alias \mapsto freeaddr], \mathcal{H}^\#[freeaddr \mapsto (\text{Module}, freeaddr, mod)])
\end{aligned}$$

3.5 Type Annotations

Notice that we have not yet talked about 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:

```

1 from mypreciouslib import amatrix
2 from pytropos.hints.numpy import NdArray
3
4 mat: NdArray[1,12,6,7] = np.array(amatrix)
5 newmat = (mat + 12).reshape((12*6, -1)) # newmat has shape (12*6,7)
6 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:

$$\begin{aligned}
\mathbb{S}[\text{AnnAssign}(var, hint, val)](\mathcal{G}, \mathcal{H}) &:= \\
\text{let } (\mathcal{G}_1, \mathcal{H}_1, ass) &:= \mathbb{E}[var](\mathcal{G}, \mathcal{H}) \\
(\mathcal{G}_2, \mathcal{H}_2, rightval) &:= \mathbb{E}[val](\mathcal{G}_1, \mathcal{H}_1) \\
compval &:= \text{if } is_value(rightval) \text{ then } val \text{ else } \langle \text{Execution Halt} \rangle \\
(\mathcal{G}_3, \mathcal{H}_3, hinteval) &:= \mathbb{E}[hint](\mathcal{G}_2, \mathcal{H}_2) \\
hintval &:= \text{if } is_value(rightval) \text{ then } val \text{ else } \langle \text{Execution Halt} \rangle \\
rval &:= \text{if } hintval < compval \text{ then } hintval \text{ else } compval \\
\text{in match } ass \text{ in} & \\
\text{case } id : \text{Val} \rightarrow &(\mathcal{G}_3[ass \mapsto rval], \mathcal{H}_3) \\
\text{case } ((t, addr, o) : \text{Object}, ('index', val : \text{Val})) \rightarrow & \\
\text{let } setindex &:= get_prim_set_index(t) \\
\text{in } setindex &(\mathcal{G}_3, \mathcal{H}_3, o, val, addr, rval) \\
\text{case otherwise} &\rightarrow \langle \text{Execution Halt} \rangle
\end{aligned}$$

Notice that we determine the “precision” of a hint with respect to the variable with the comparison between values, the order operation for the Abstract Domain **Val**!

Further details on the concrete and abstract semantics are given in Appendix [A](#).

Chapter 4

Pytropos (Analysis Implementation)

Pytropos¹² is the Abstract Interpreter implemented in this work. The interpreter follows the rules exposed in the previous chapter.

We explore how is Pytropos built, a brief history and some internal details.

4.1 The big picture

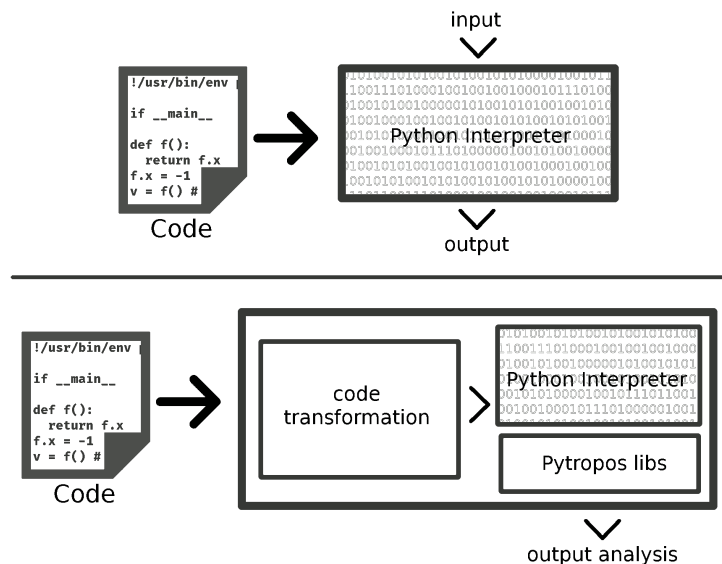


FIGURE 4.1: Up: black box step by step of executing a piece of code in Python (CPython).
Down: Pytropos extension of Python. Pytropos transforms the code prior to run it by CPython.
The transformation wraps the code to be run with the help of a custom library.

Pytropos works in a similar way as any other Python interpreter does. Pytropos reads code and executes it sentence by sentence. Its main difference to CPython is that it does not transform code into bytecode but it

¹There 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 Greek mythology, Atropos was one of the three goddesses of fate who decided on the lives of humans, she was the goddess of death and the one who cut the thread of life.

²Homepage: <https://github.com/helq/pytropos>

wraps the code before it is transformed into CPython bytecode. The code is wrapped to use a library that implements the semantics of the Abstract Interpreter. See Figure 4.1.

Similar to how Python works, Pytropos not only “runs” 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

The traditional methodology to build an interpreter is to code the parser and the semantics of the language in one big program. For a language like Python, building an interpreter from scratch would require considerable effort given the large amount of characteristics Python comes with. A “complete” Python implementation would require to implement—among other things—memory management, garbage collection, function calling and return, dynamic type analysis, exception handling.

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

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

Notice 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 import typeops as to
2 a = to.add(int, int)
3 b = to.add(a, str)
4 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 Type 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 errors³.

The main advantage of translating code into wrapped code is the reuse of infrastructure. One does not need to write all the infrastructure 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.

³This strategy has been applied in the past for similar purposes, mainly to reuse infrastructure. It was used by Lauko, Ročkai, and Barnat (2018) for symbolic computation of LLVM bytecode.

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 accesses must also be wrapped, otherwise the execution would stop if one finds an undefined variable. Because of all of this wrapping, the transformation does not produce a simple, human-readable output.

To show, how currently Pytropos wraps and transforms the code consider the code below from the piece of code above:

```

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

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 of 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 tool⁴. Pytropos is meant to be an aid for developers when working with tensors.

Based on that, I present the main assumptions taken into account at the design stage 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 their actual content.

⁴I use MyPy and flake8 in every project and I am thankful for the years of effort put into these amazing tools. Thank you, guys!

- 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 more precise than the value that Pytropos has already inferred the inferred value will not be changed.

4.4 Details about 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 ($\mathcal{H}^\#$) but rather, I 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 `PythonValue`s directly (again, because CPython manages the heap not Pytropos). 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 and operations, and it can point to any `PythonValue`.

The class `PythonValue` implements the `Val#` Abstract Domain. A `PythonValue` is a wrapper around either an `AbstractValue` or an `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.

`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` an 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`.

Figure 4.2 presents the class diagram showing the relationships between the different components in Pytropos.

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

The implementation of the Abstract Interpreter also includes a class log that stores all warnings generated in the abstract interpretation of the code.

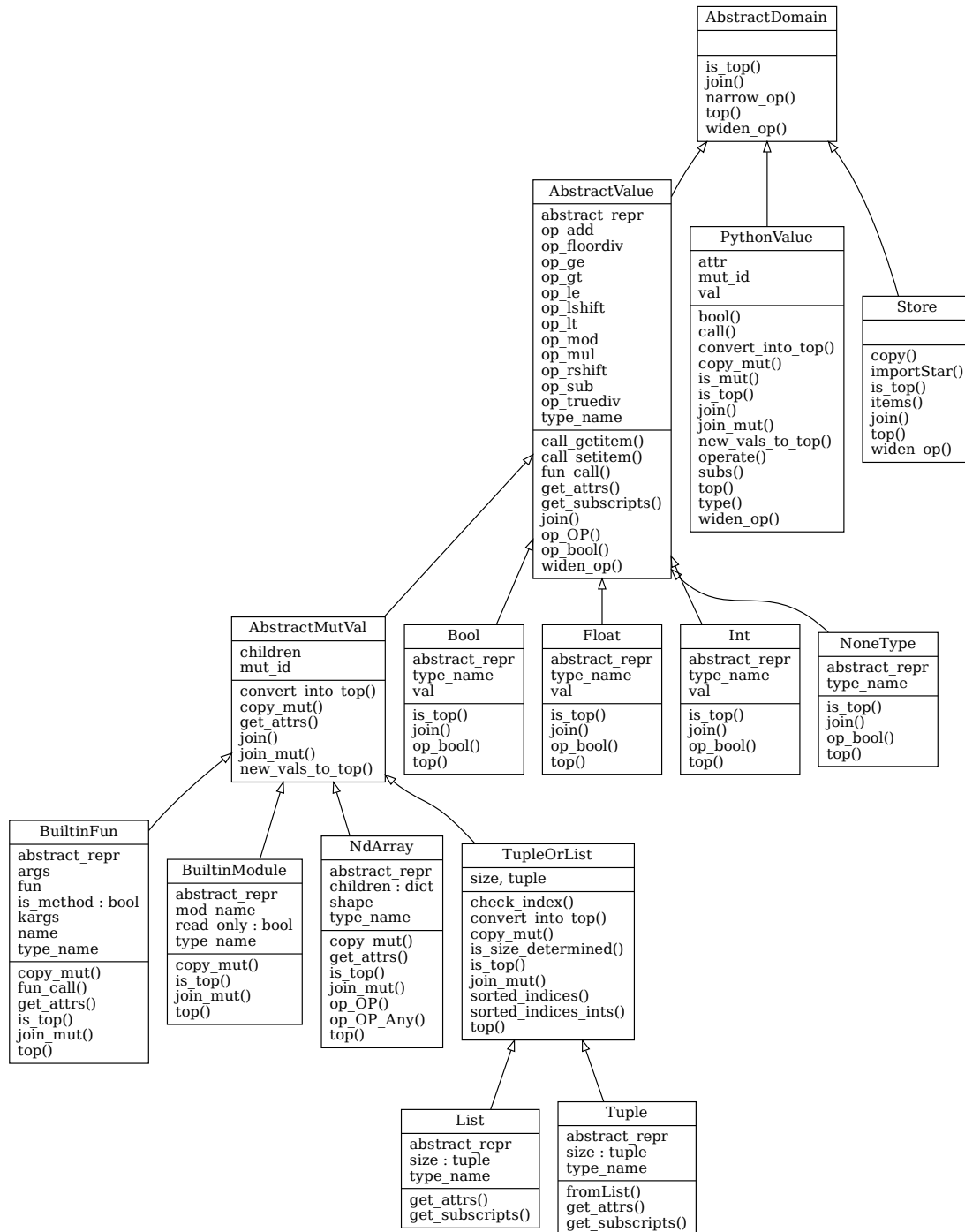


FIGURE 4.2: Class diagram for Pytropos, the implementation of the Abstract Interpreter.

Chapter 5

Validation and Discussion

In this chapter, we explore the capabilities of Pytropos, the implementation of the proposed abstract interpreter. The chapter is divided into two parts: how was the code tested and a discussion on the results of the tests.

5.1 Validation

We created two types of tests: unit regression tests and property-based tests. Unit tests fulfill two purposes: they are a guide of what the abstract interpreter should do and they act as regression checks to prevent reintroducing bugs. Slightly more challenging to write are property-based tests. They allow us to make sure that Pytropos complies to the specification and to CPython behaviour.

Comparing against the official regression battery of tests for CPython is left for future work as the abstract interpreter still is not ripe enough. The abstract interpreter is still missing a great number of built-in functions and classes.

5.1.1 Unit Tests

There are a total of 74 unit tests checking various parts of the Abstract Interpreter. The tests can be categorised into:

- `if` branching
- `while` looping
- `list` s and `tuple` s
- NumPy array operations
- Type annotations
- Miscellaneous

Table 5.1 shows the results of evaluating in Pytropos all unit tests. There are three cases for the evaluation of a unit test:

- ✓: The test ran as it was expected, i.e. Pytropos catch all expected errors and computed all values correctly.
- ✗: The test did not run as expected, i.e. Pytropos did not catch all expected errors or computed some value incorrectly.
- ✚: It was not possible to run the test because some method, function, or operation has not yet been implemented.

The tests check not only for what Pytropos is able to check now but what it should be able if implemented the whole semantics described in Appendix A.

TABLE 5.1: Unit test coverage on the different capabilities of the abstract interpreter. ✓ indicates the number of tests that passed as expected. ✗ indicates the number of tests that failed to run as expected. + indicates the number of tests that failed to run due to an incomplete implementation.

Category	Total lines	Tests	✓	✗	+	Total coverage
if branching	133	16	14	2	0	87.5%
while looping	99	11	10	0	1	90.9%
list s and tuple s	74	11	9	1	1	81.8%
NumPy array operations	315	25	20	3	2	80%
Type annotations	26	4	1	0	3	25%
Miscellaneous	32	7	5	0	2	71.4%

As it can be seen in Table 5.1, Pytropolis passes most tests as expected, i.e. Pytropolis conforms to our definition of the Abstract Interpreter catching all expected errors and calculating correctly the values of the variables. The tests that Pytropolis does not pass are due to subtle semantic rules that have not yet been implemented or a lack in the support of built-in values. Pytropolis errors come from too much overapproximation and never as a result of an underapproximation.

5.1.2 Property-based Tests

Property-based testing consists in writing a property that a piece of code should fulfill. For example, a property of lists is that adding a new element to a list of size `n` gives back a list of size `n+1`. Property-based testing tools try to find a counterexample for a given property. We use property-based testing to check for compliance of built-in operations in Pytropolis against CPython.

There are a total of 20 property-based tests. We use Hypothesis¹ to find counterexamples. For each test, Hypothesis generates 100 inputs following a predefined strategy which looks for common inputs that tend to break code and random inputs. Property-based testing proved to be extremely helpful at the initial stages of Pytropolis development, as Hypothesis was able to show several weak spots in the implementation.

Many operations between Python values throw exceptions, e.g. `3 / 0` throws a zero-division exception. Some of the values that trigger exceptions should be known to every programmer, but others have been quite surprising as they have appeared. Thanks to Property-based tests, Pytropolis primitive values have been extensively tested. The following are some of the exceptions that Hypothesis helped us find:

- Zero-division exception on expressions like `5 % 0` or `5 % False`
- Value exception on expressions like `5 « -2`
- Excessive High-memory consumption on expressions like `5 « 10**10`
- Overflow exception on expressions like `10**209 + 2.0`

5.2 Discussion

In this section, we will show some positive and negative unit tests. Through them, we will explain what is Pytropolis capable of doing and when does it fail.

¹Hypothesis is a testing tool designed to strategically generate input examples in an attempt to disprove a property about the code. Tests in Hypothesis are properties of what the developer thinks the code should do. Homepage: <https://hypothesis.works>

5.2.1 Pytropos capabilities

The following are some tests where Pytropos runs as expected and it is able to compute all values correctly:

- Pytropos is able to join two states with primitive variables properly:

<pre> 1 a = 5 2 3 if True: 4 c = 2 5 else: 6 c = 2.0 7 8 d += 1 </pre>	<pre> 1 a = 5 2 3 if False: 4 c = 2 5 else: 6 c = 2.0 7 8 d += 1 </pre>	<pre> 1 a = 5 2 3 if d: 4 c = 2 5 else: 6 c = 2.0 7 8 d += 1 </pre>
--	---	---

In the first two examples, Pytropos selects correctly which branch to execute and warns the user of the use of an undeclared variable, `d`. All undeclared variables are set by default to `Undefined`. If an undeclared variable is used, it is set to \top_{Val} .

In the third example, Pytropos warns the user of the undeclared use of the variable `d` and chooses to execute both branches separately with a copy each of the state of the program. The new state of the program is the result of joining the two final states of running each branch.

The final states for the examples are:

<code>'a' ↦ 5</code>	<code>'a' ↦ 5</code>	<code>'a' ↦ 5</code>
<code>'c' ↦ 2</code>	<code>'c' ↦ 2.0</code>	<code>'c' ↦ \top_{PrimVal}</code>
<code>'d' ↦ \top_{Val}</code>	<code>'d' ↦ \top_{Val}</code>	<code>'d' ↦ \top_{Val}</code>
<code>_ ↦ Undefined</code>	<code>_ ↦ Undefined</code>	<code>_ ↦ Undefined</code>

- Pytropos can join two states with non-primitive values, e.g. lists.

<pre> 1 from importantlib import either 2 3 either: int = either 4 5 a = [1] 6 a[0] = a 7 8 if either > 20: 9 a.append(2) 10 else: 11 a.append(3) 12 a[1] -= 1 </pre>
--

Operating with a variable imported from a library is the same as to work with an undefined variable. The imported variable `either` is set to \top_{Val} .

The type annotation on `either: int = either` tells Pytropos that `either` is an `int` value. The result of computing the expression `either > 20` is \top_{Bool} not a \top_{Val} . Even though \top_{Bool} is more precise than \top_{Val} ($\top_{\text{Bool}} < \top_{\text{Val}}$), it is not precise enough to tell us which branch will be executed (\top_{Bool} tells us that the variable may either be `True` or `False`). Thus, both branches must be executed and joined.

`a` contains two elements after running `a.append(2)`: a reference to itself and the integer 2. And `a` contains two elements after running `a.append(3); a[1] -= 1`: a reference to itself and the integer 2. Both states are the same, therefore their joined state is the same. The final state of the program is:

$$\mathcal{G}(i) = \begin{cases} 0 & \text{if } i = \text{'either' } \\ 1 & \text{if } i = \text{'a' } \\ \text{Undefined} & \text{otherwise} \end{cases}$$

$$\mathcal{H}(n) = \begin{cases} \top_{\text{Int}} & \text{if } n = 0 \\ (\text{List}, 1, l) & \text{if } n = 1 \\ 2 & \text{if } n = 2 \\ 2 & \text{if } n = 3 \\ \text{Undefined} & \text{otherwise} \end{cases}$$

$$l(i) = \begin{cases} 2 & \text{if } i = \text{'size' } \\ 1 & \text{if } i = (\text{'index'}, 0) \\ 3 & \text{if } i = (\text{'index'}, 1) \\ \text{Undefined} & \text{otherwise} \end{cases}$$

- Pytropos can apply the widen operator after trying to run a piece of code a couple of times.

```

1 i = 0
2 j = 0
3
4 while i < 10:
5     j += i
6     i += 1

```

```

1 i = 0
2 j = 0
3
4 while True:
5     j += i
6     i += 1

```

```

1 i = 0
2 j = 0
3
4 while i < n:
5     j += i
6     i += 1

```

Anytime Pytropos is asked to execute a loop, it tries to run the loop like any other interpreter, i.e. Pytropos will run the body of the loop until the condition becomes false. For example, in the first piece of code the condition `i < 10` becomes false after 10 iterations, and the final state of the program is reached (`i` and `j` have values 10 and 45, respectively).

To prevent Pytropos from running forever any loop is stopped after a set number of iterations and it is assumed that the truth value cannot be determined. If it is not possible to determine the truth value of the condition, Pytropos will loop once on a copy state and it will apply the widen operation on the two states. Abstract Interpretation warranties that the repeated application of the widen operation on an increasing sequence (loop application) will eventually stop and find a fix-point. The second and third example arrive at the same state, `i` and `j` have the value \top_{Int} , albeit after different executions are made. In the second example, Pytropos executes the loop until it reaches the maximum number of executions allowed, then it executes the body of the loop in a copy of the state and applies the widening operator on the two states. In the third example, no execution prior to applying the widening operation is performed as the truth value of `i < n` cannot be determined as `n` has not been defined.

- Pytropos is able to determine appropriately the shapes of NumPy arrays defined from lists, tuples and NumPy arrays.

```

1 import numpy as np
2
3 m = n = _
4
5 _ = np.array(3).shape           # must be ()
6 __ = np.array([3]).shape        # must be (1,)
7 a = np.array([[2,3],[3,4,5]]).shape # must be (2,)
8 # show_store(a)

```

```

9  b_ = np.array([[2,3,4],[3,4,5]])          # must be array(shape=(2,3))
10 b = b_.shape                             # must be (2,3)
11 c = np.array([[2,3],7]).shape             # must be (2,)
12 d = np.array([[2,3,1],[2,2,2]]).shape     # must be (2,3)
13 e = np.array([m,(2,2,2)]).shape           # must be (2,...)
14 f = np.array([m,[1,2],[2,2,2]]).shape     # must be (3,)
15 g = np.array([m,n]).shape                 # must be (2,...)
16 ls = [[1,2,3,4,5],[3,4,0,0,1]]
17 h = np.array([ls,ls,ls]).shape             # must be (3, 2, 5)
18 i = np.array([ls,ls,n]).shape              # must be (3, ...)
19 j = np.array([ls,[ls[0], m],n]).shape      # must be (3, ...)
20 k = np.array([ls,[ls[0], m],n])).shape    # must be (1, 3, ...)
21 l = np.array(b_).shape                    # must be (2,3)

```

This test case shows how Pytropos is able to calculate the correct shape for both: values that have a determined shape like lists of numbers, and \top_{val} values which could have any shape.

- Pytropos detects correctly when an operation between two tensors is incorrect, and it can calculate the correct resulting shapes of a successful operation application.

```

1  import numpy as np
2
3  a = (np.zeros( (2, 1, 4)) + np.zeros( (3,1)) ).shape # must be (2, 3, 4)
4  b = (np.zeros( (2, 2, 4)) + np.zeros( (3,1)) ).shape # broadcasting error
5  c = (np.zeros( (2, 2, 4)) - 3 ).shape # must be (2, 2, 4)
6  d = (np.zeros( (7, 1, 8)) / [7] ).shape # must be (7, 1, 8)
7  e = (np.zeros( (2, 2, 4)) * [7,3] ).shape # broadcasting error
8  f = ( ([2],[3],[6]) % np.ndarray( (5,1,1)) ).shape # must be (5, 3, 1)

```

Notice that Pytropos computes the shape of non-arrays like `3` or `[7,3]` because NumPy converts any non-array into array before computing between them, e.g. the expression `np.zeros((7, 1, 8)) / [7]` is the same as computing `np.zeros((7, 1, 8)) / np.zeros((1,))`.

- Pytropos uses type annotations when it is not able to compute the shape of a tensor.

```

1  import numpy as np
2  import somelib
3
4  from pytropos.hints.numpy import NdArray
5
6  a: NdArray[2,3,4] = np.array(somelib.val) # a should be array(shape=(2,3,4))
7  b: NdArray[1,2] = somelib.numpyval()     # b should be array(shape=(1,2))
8  c = [[6], [8]]
9  c = np.array(c)                          # c should be array(shape=(2,1))
10 d = b + c                                # d should be array(shape=(2,2))
11 e: NdArray[2, 5] = c                     # Error. e should be array(shape=(2,1))
12 f: NdArray[()] = np.array(2)              # Everything ok. f is array(shape=())
13 g: NdArray[int,2] = somelib.numpyval()    # g should be array(shape=(int?,2))

```

```

14 h: ndarray[6,2] = np.array(g)           # h should be array(shape=(6,2))
15 i: ndarray[10] = somelib.x()           # i should be array(shape=(10,))

```

Type annotations can only improve the current computed value. If a faulty type annotation is given, Pytropos will signal the user. For example, the type annotation in the assignment `e: ndarray[2, 5] = c` is incorrect as the shape of `c` is computed to be `(2,1)` but the annotation says `(2,5)`.

5.2.2 Pytropos failures

The following are all the tests where Pytropos does not work as intended. It either fails to compute the values correctly or fails to run the code.

- Pytropos overapproximates some join operations between states. When Pytropos joins two states that are not precisely equal, it may collapse a variable into a \top . This is not always as precise as one may expect. For example, consider:

```

1 from somelib import either
2
3 if either:
4     a = []
5     b = a
6 else:
7     a = []
8     b = []

```

In the `then` branch a single list is created and both variables point to it. In the `else` branch two lists are created, one for each variable. The state that better captures the join of both states is:

$$\begin{aligned}
 'a' &\mapsto \top_{\text{List}} \\
 'b' &\mapsto \top_{\text{List}} \\
 _ &\mapsto \text{Undefined}
 \end{aligned}$$

But Pytropos calculates the slightly more general case:

$$\begin{aligned}
 'a' &\mapsto \top_{\text{Val}} \\
 'b' &\mapsto \top_{\text{Val}} \\
 _ &\mapsto \text{Undefined}
 \end{aligned}$$

- Pytropos is not yet able to apply the widening operator on two states with non-primitive values.

```

1 from somewhere import something
2
3 a = []
4 a.append(3)
5 i = a[0]
6 while i < 10:

```



```

7   if something:
8       a.append(i)
9   else:
10      a.append(i+1)

```

In the example, Pytropos runs the loop until it reaches the iteration limit. In which point, it runs the loop one more time in a state copy and applies the widening operator on the states. Pytropos fails to run any further as the widening operator has not yet been implemented for lists.

- Pytropos is in active development, thus it lacks support to many primitives.

```

1  import numpy as np
2  from pytropos.hints.numpy import NdArray
3  from something import Top
4
5  a: NdArray[3, float] = np.arange(Top).reshape((3, Top))

```

In the example, the expected result is a warning explaining that one of the dimensions of an array cannot be a floating number. All dimensions of an array must be natural numbers. Pytropos flags the code as faulty because `float` has not been defined.


5.2.3 Using Pytropos as a linter

Any time Pytropos detects an error in the interpretation, Pytropos logs the error in a structure to be later printed out to the user.

Writing a plugin to integrate Pytropos to any IDE is straightforward. Pytropos outputs into console all errors in a standard format. Pytropos outputs all warnings in the format:

```
file:col:line: ERRORCODE: Warning message
```

A plugin in vimscript was written for Pytropos to use Pytropos as a linter for the text editor (Neo)Vim. As can be seen in Figure 5.1, the developer can get feedback on their code on real-time.



```

16
17 # Both arrays are 2-D
W 18 g = zeros((2,3)).dot([[1,3]]) > W503 ValueError: shapes (2, 3) and (1, 2) not aligned: 3 (dim 1) != 1 (dim 0)
19 h = zeros((12,3)).dot([[1],[2],[3]])
20
21

```

FIGURE 5.1: Example of a warning message displayed in (Neo)Vim as a result of checking the code with Pytropos.

The required steps to use Pytropos as a linter in (Neo)Vim are:

- Install Pytropos and make sure it can be run from console.
- Install the plugin Neomake. Homepage: <https://github.com/neomake/neomake>.
- Tell Neomake how to interpret the Pytropos output by adding the following lines to the configuration file `.vimrc`:

```

function! PostProcessingPytropos(entry)
    " Pytropos uses 0-indexed columns, and Neovim uses 1-indexed
    let a:entry.col += 1

    if a:entry.text =~# '^E' || a:entry.text =~# '^SyntaxError'
        let a:entry.type = 'E'
    elseif a:entry.text =~# '^W'
        let a:entry.type = 'W'
    elseif a:entry.text =~# '^F'
        let a:entry.type = 'F'
    else
        let a:entry.type = 'A'
    endif
endfunction

let g:neomake_python_pytropos_maker = {
    \ 'exe': 'pytropos', "Change if Pytropos is called differently on your machine
    \ 'args': [],
    \ 'errorformat':
        \ '%f:%l:%c: %m',
    \ 'postprocess': [
        \ function('PostProcessingPytropos'),
        \ ]
    \ }

let g:neomake_python_enabled_makers += ['pytropos']

```

5.2.4 Summary

Pytropos is still at a very early stage of development. The repertoire of Python characteristics implemented in it is still small: no support for **for** statement, custom classes and objects, exception handling and the lack of built-in definitions. Nonetheless, Pytropos can check for errors on operations between tensors and can be extended to work with more libraries.

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 and can handle tricky or complex methods as the function `array` from NumPy. The NumPy function `array` can evaluate almost anything as an array. Pytropos is able to calculate correctly the value of primitive Python values like integers and floats. It can join two values and states of appropriately. It is able to apply the widen operator in certain conditions.

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 them 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 are given.

6.1 Static Analysis in Python

In Table 6.1, a list of all analysis tools found in the literature and libraries is given. The usage of a tool refers to how is the tool used by developers: embedded in the python interpreter, used as a linter, or as a library that runs with the code to analyse. The Analysis base of a tool refers to which is the theory behind the tool, how it works underneath.

Cannon (2005) was the first (to our knowledge) to try to statically infer and type check code in Python. His idea was to infer variables' types before executing the code and use this knowledge to speed up the execution. He acknowledged that the nature of Python makes type inference to be too weak. Thus, the complexity that the project brought did not improve performance in equal measure and his work was never added to Python. This work was prior to Siek and Taha (2006) with Gradual Types, where the developer can help the type checking algorithm to find and assert better types.

All linters' goal is to aid the developer by flagging code as faulty. What means for a piece of code to be faulty varies from tool to tool. Pep8 checks for syntactic deviations of the style guide, Pep 8¹. Pyflakes, Pylint and PyChecker define a variety of checks for common scenarios where a piece of code is known to fail, e.g. undefined variables. MyPy, PyType and Pyre focus on type checking optionally, type-annotated code. Nagini focuses on verifying pieces of Python code using the mature Viper (Müller, Schwerhoff, & Summers, 2016) infrastructure. Pytropos objective is different from all the other available linters: Value Analysis.

Our work is closest to Fromherz et al. (2018) as they focus on Value Analysis as well. Fromherz et. al focus is on building an Abstract Interpreter to verify Python code. Our focus is not in verification but in the shape of Tensors. Their Abstract Interpreter incorporates more capabilities than we do: exception handling and support for breaking flow instructions. Unfortunately, they have not yet made their implementation public.

6.2 Tensor Shape Analysis

6.2.1 Solutions in other languages

We may think that 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

¹<https://www.python.org/dev/peps/pep-0008/>

TABLE 6.1: Analysis tools for Python.

Tool Name / Source	Usage	Analysis base	Purpose
Cannon (2005)	Embedded	Type analysis	Typechecking
Pep8 ^o	Linter	N/A	Code style analysis
Pyflakes ¹	Linter	N/A	Various checks
Pylint (Thenault et al., 2006)	Linter	N/A	Various checks
PyChecker (Norwitz, n.d.)	Linter	N/A	Various checks
MyPy (Lehtosalo et al., 2016)	Linter	N/A	Type checking
Enforce ²⁺ (defunct)	Library	Gradual Type Analysis	Type checking
Sagitta ³⁺ (defunct)	Library	Gradual Type Analysis	Type checking
StyPy (Ortin, Perez-Schofield, & Redondo, 2015)	Linter	Type analysis	Type checking
Lyra ⁴	Library	Novel	Type checking
Pytropos ⁵ (This Work)	Library	Abstract Interpretation	Various analyses (including Value Analysis)
PyType ⁶	Linter	Abstract Interpretation	Value Analysis (specialised on tensor shapes)
ICBD ⁷ (defunct)	Linter	N/A	Type checking and inference
Pyre ⁸	Linter	N/A	Type checking and inference
Nagini ⁹ (Eilers & Müller, 2018)	Linter	Abstract Interpretation	Type checking
Typete ¹⁰ (Hassan, Urban, Eilers, & Müller, 2018)	Library	SMT (Viper)	Verifier
Fromherz, Ouadjaout, and Miné (2018) [*]	N/A	Abstract Interpretation	Type inference
Monat (2018) [*]	N/A	Abstract Interpretation	Type Analysis
PyAnnotate ⁺¹¹	Library	Gradual Type Analysis	Type inference
MonkeyType ⁺¹²	Library	Gradual Type Analysis	Type inference

⁺ It is not an static analysis. It is a dynamic analysis.

^{*} Tool has not been made public.

^o Homepage: <https://pep8.readthedocs.io/>

¹ Homepage: <https://pylint.org/project/pyflakes/>

² Homepage: <https://github.com/RussBaz/enforce>

³ Homepage: <https://github.com/peterhl/sagitta>

⁴ Homepage: <https://github.com/caterinaurban/Lyra>

⁵ Homepage: <https://github.com/helq/pytropos>

⁶ Homepage: <https://github.com/google/pytype>

⁷ Homepage: <https://github.com/kmod/icbd>

⁸ Homepage: <https://github.com/facebook/pyre-check>

⁹ Homepage: <https://github.com/marcoeilers/nagini>

¹⁰ Homepage: <https://github.com/caterinaurban/Typete>

¹¹ Homepage: <https://github.com/dropbox/pyannotate>

¹² Homepage: <https://github.com/Instagram/MonkeyType>

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 exist. 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-detyped` written by Cruz-Camacho and Bowen (2018). The library is written as a wrapper around the library port of TensorFlow for Haskell. `tensorflow-haskell-detyped` enforces at the type level how and which results an operation between compatible tensors are to be performed.

A different path, not yet explored, 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-detyped`).

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; Trojahnner & Grelck, 2009). All solutions, though, are theoretical, they propose type systems which, if they 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 literature²:

- Trojahnner 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 on 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”.

²As 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.

- 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 dependent types. The special keyword of this work is "tensor manipulation language".

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this work, we have presented the definition and implementation of an Abstract Interpreter for Python focused on the analysis of tensor shapes.

We showed how can Abstract Interpretation be applied to a dynamically typed programming language like Python with the goal of defining a Static Value Analysis. To make this possible, first, we specified small-step (concrete) semantics for Python taking into account Python dynamically typed variables and aliasing. To build the Abstract Interpreter we defined an apt Abstract Domain for all values in the language, an Abstract Domain for a state of a program with aliasing, and abstract semantics. Our approach to build the Abstract Interpreter is easily extendable to check for more than built-in values as we showed by extending the syntax and semantics of Python to handle the shape of NumPy arrays (tensors). The Abstract Interpreter is able to use type annotations from the user to get more precision on the values it is able to analyse.

We presented a working implementation of the Abstract Interpreter, Pytropos, which is able to analyse the shape of tensors in a variety of scenarios. The interpreter can work as a linter for IDEs catching potential errors as the developer codes.

In brief, the work done includes:

- A specification of a subset of Python 3.6 (Syntax and small-step semantics).
- 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, Pytropos¹.
- An application for the Abstract Interpreter to statically analyse the shapes of tensors and tensor operations.
- A series of tests showcasing the abilities of the Abstract Interpreter and its failures.
- A way for developers to annotate code to improve the accuracy of the Abstract Interpreter.

The presented Python formalisation is easily extendible to calculate the value of a library-defined class. NumPy arrays were added to the formalisation to show how to analyse code the shape of tensors.

The abstract semantics, the semantics of the Abstract Interpreter, are easy to develop. It was shown how to get from the concrete semantics (small-step semantics) to the abstract semantics.

Testing showed that the Pytropos, the implementation of the Abstract Interpreter, is able to check many common shape mismatches. The biggest problem of the implementation is on the lack of support for built-in functions and values.

¹Available at: <https://github.com/helq/pytropos>

When the Abstract Interpreter cannot compute the correct tensor shape, the user may help the Abstract Interpreter with Type Annotations. If the user gives a type annotation that does not improve the precision of the computed value, the interpreter will warn the user of his mistake.

7.2 Future Work

Python is a huge and rich language. The amount of characteristics that Python has exceeds by far what a single human can implement in the span of a master thesis.

Much work is left to improve Pytropos so that it can be used by the regular developer. We propose the following roadmap to continue building the Abstract Interpreter:

- Extend Pytropos to include Exception handling. A similar approach to that of Fromherz et al. (2018) could be a good starting point.
- Improve how copying and joining stores (states of the program) are done. The join operation between stores is very, very costly. Walking through the graphs becomes prohibitively expensive as the program to analyse grows in number of `if` and `while` sentences. This associated cost could be reduced if only “diff”s of stores were used. One way to do this is by using immutable structures for all values in the implementation. Using immutable structures would require the explicit implementation of the heap and garbage collector.
- Extend Pytropos to handle breaking control statements (`continue` , `break` and `return`). Fromherz et al. (2018) also present a way to handle breaking control statement in Python.
- Extend the global scope to handle local and nonlocal scopes. The scope rules of Python are mildly complex with four different variable scopes: global, local, nonlocal, and class. Something to take into account is the ability of CPython to statically analyse the use of local and non-local variables before interpreting the code.
- Extend `Val#` with user-defined functions and objects. 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 with objects is implementing the MRO rules in charge of how to inheritance work in 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 presented some concrete and abstract semantics for Python but there was never a proof of their correctness. We consider the following to be the problems to solve to prove the formalisation correct:

- Give a complete and through formal definition for a subset of Python in a proof-assistant language such as Agda, Idris, Coq, and Isabelle/HOL.
- Define in the same proof-assistance language the Abstract Domain, the properties it must follow, and the abstract semantics.
- Prove that the abstract semantics are in fact consistent with the concrete semantics and abstract domain.

Appendix A

Python Static Analysis based on Abstract Interpretation

This appendix is divided into two parts: first, we present a reduced set of the Python syntax together with a (partial) operational semantics; second, we show the use of the formal semantic definition to define the abstract interpretation solution we implemented.

This chapter is meant to explain the theory behind the Abstract Interpreter implemented. In Chapter 4 an explanation on the implementation details is given.

A.1 Python (reduced) syntax and semantics

Python has no official small-step (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 small-step semantics defined for Python (Politz et al., 2013; Fromherz et al., 2018; Guth, 2013; Ranson et al., 2008). In this work, we define yet one more time small-step semantics for Python. We decided to define our small-step semantics for Python because of our specific needs.

Our semantics definition resembles the most that of Fromherz et al. (2018). In fact, our definition is loosely based on Fromherz et al. (2018). As Fromherz et al., we define each expression and statement step as a function from states to states of the program, but opposed to them we do not include exception handling and control break statements. Our definition 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 our definition:

```
1 a = []  
2 b = a.append  
3 b(3)  
4 a.append(2.0)  
5 print(a) # prints: [3, 2.0]
```

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

A.1.1 (Reduced) Syntax

Our simplified version of Python's Syntax can be seen in Figure A.1. Modified from AST's syntax below from Figure A.2.

$expr$	$:=$	$i \in \mathbb{Z} \mid j \in \text{float} \mid \text{True} \mid \text{False} \mid \text{None}$	primitive values
		$identifier$	variable name
		$expr \text{ op } expr \mid expr \text{ cmpop } expr$	binary operations
		$expr(expr \text{ * })$	function call
		$expr.identifier$	attribute access
		$expr[expr]$	subscript access
		$[expr \text{ * }]$	lists
		$(expr \text{ * })$	tuples
$stmt$	$:=$	$\text{del } expr$	delete
		$expr = expr$	assignment
		$expr \text{ op } = expr$	augmented assignment
		$expr: expr = expr$	type annotation
		$\text{while } expr: stmt^+$	loop
		$\text{if } expr: stmt^+ \text{ else: } stmt^+$	if-else
		$\text{import } alias^+ \mid \text{from } identifier \text{ import } alias^+$	import library
		$expr$	
op	$:=$	$+ \mid - \mid * \mid / \mid \% \mid ** \mid \ll \mid \gg \mid //$	
$cmpop$	$:=$	$< \mid <= \mid > \mid >=$	
$alias$	$:=$	$identifier \mid identifier \text{ as } identifier$	

FIGURE A.1: Reduced Python syntax used throughout this work.

The syntax from Figure A.1 is a subset of the Python 3.6 syntax¹. Note that CPython does not directly interpret code written in this syntax. The usual steps of lexing and parsing into a more explicit representation are necessary for a program to be executed by CPython. We will define the semantics of the language over the parser's syntax, see Figure A.2 and not over the aforementioned syntax. The reasoning behind this change is due to the parser's syntax readiness to be analysed opposed to the original syntax.

Code written in the subset of Python (Figure A.1) gets translated into the parser's representation (Figure A.2), over which we define the semantics of the language. We will not describe the process of translation (parsing), but we will explore some examples to show why the parser's syntax aids into the definition of the small-step semantics:

- `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.
- `a = 3` gets translated into `Assign(Name(a, Store), Int(3))`. Notice the `Store` context, it tells us that we want to modify the variable `a`, i.e. we want a reference to the variable.
- `del a` gets translated into `Delete(Name(a, Del))`. Notice the `Del` context, it tells us that we will where the object location in the heap.
- `a.b[3] + b.c` gets translated into

¹Modified from the Python 3.6 syntax found at https://github.com/python/typed_ast/blob/89242344f18f94dc109823c0732325033264e22b/ast3/Parser/Python.asdl

$expr$	$:=$	$Int(i) \text{ for } i \in \mathbb{Z} \mid Float(j) \text{ for } j \in \text{float}$	
		$True \mid False \mid None$	primitive values
		$Name(identifier, expr_context)$	variable name
		$BinOp(op, expr, expr)$	
		$Compare(expr, cmpop, expr)$	binary operations
		$Call(expr, expr^*)$	function call
		$Attribute(expr, identifier)$	attribute access
		$Subscript(expr, expr, expr_context)$	subscript access
		$List(expr^*)$	lists
		$Tuple(expr^*)$	tuples
$stmt$	$:=$	$Delete(expr)$	delete
		$Assign(expr, expr)$	assignment
		$AugAssign(expr, op, expr)$	augmented assignment
		$AnnAssign(expr, expr, expr)$	type annotation
		$While(expr, stmt^+)$	loop
		$If(expr, stmt^+, stmt^+)$	if
		$Import(alias^+)$	import library
		$ImportFrom(identifier, alias^+)$	import from
		$Expr(expr)$	
$expr_context$	$:=$	$Load \mid Store \mid Del$	
op	$:=$	$Add \mid Sub \mid Mult \mid Div \mid Mod \mid Pow$	
		$LShift \mid RShift \mid FloorDiv$	
$cmpop$	$:=$	$Lt \mid LtE \mid Gt \mid GtE$	
$alias$	$:=$	$identifier \mid (identifier, identifier)$	

FIGURE A.2: Python syntax used to define the semantics of the Abstract Interpreter.

```

1 BinOp(Add,
2     Subscript(Attribute(Name(a, Load), b), Int(3), Load),
3     Attribute(Name(b, Load), c)
4 )

```

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

```

1 Assign(
2     Subscript(Attribute(Name(a, Load), b), Int(3), Store),
3     Int(3)
4 )

```

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.

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

```

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

```

Notice how we only get the `Del` context for the subscript and not for anything else, as we only want where and who belongs to the subscript.

The semantics of a `del` sentence require us to know where the identifier or attribute is located (in an object or the store). Assigning a variable requires us to know where to put a variable. Accessing to inexistent attribute (`class A(): ...; a = A(); a.length # error: attribute unknown`) is 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 these three different statement-dependent expressions.

Despite the inability to define a custom function or class in the language, we want to be able to call a function and access to objects attributes. That is why we have added them into the syntax to handle.

We have found that even though the amount of characteristics we support 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

For us, there are two types of Python values:

- Primitive values, **PrimVal**, which include integers (`int` s), Floating point numbers (`float` s), 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 it is located, and a “dictionary” pointing to other values.
- Functions: **<prim-callable>** and **Addr × <prim-callable>**. Primitive callables, or functions, are specific semantics rules defined directly by us. Sometimes a function may be attached to an object, in which case, the object will be passed as the parameter `self` to the function.

Lists, Tuples, built-in Modules, and built-in Classes derive from **Object**. Some mutable values may not allow any modification to their attributes, e.g. Tuples do not allow changing the content of the values they hold. The name “Mutable Values” refers to their ability to point at other values (either Primitive and Mutable).

The **state of a Python program**—called the store in Pytropos—is a tuple from $(\mathbf{Global} \times \mathbf{Heap})$ where **Global** is the “global scope” of variables, and **Heap** the heap (where all values are stored).

Putting all together we get:

```

Global  := Iden  $\rightarrow$  Addr  $\cup$  {Undefined}
Heap    := Addr  $\rightarrow$  Val  $\cup$  {Undefined}

Val     := PrimVal | Object | <prim-callable>
           | Addr  $\times$  <prim-callable> function associated to value
PrimVal :=  $i \in \mathbb{Z}$  |  $j \in \text{float}$  | True | False | None
Object  := Type  $\times$  Addr  $\times$  (Key  $\rightarrow$  Addr  $\cup$  {Undefined})
Type    := List | Tuple | Module
Key     := Iden + (string  $\times$  Val)

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

```

A **<prim-callable>** is a value that is a built-in function in Python. The special value Undefined is used to signal unassigned values in **Global** and **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 erroneous 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 number².

Notice how the index (**Key**) for the function that relates an **Object** to its attributes can be one of two things. **Key** is either a **Iden** or a tuple (string \times **Val**). The idea behind indexing an object with two separate kinds of keys is to be able to differentiate between a value that is inherent to the **Object** and others values to which the object simply points to. Consider a list, an inherent, 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 intrinsic property of the list.

A **Key** can be a tuple (string \times **Val**). There are only two “tupled” keys in the current specification, 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:

$$(\mathbf{List}, n, l)$$

where $n \in \mathbb{Z}$, and

²We are 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

$$s(i) = \begin{cases} n+1 & \text{if } i = \text{'size' } \\ n+2 & \text{if } i = (\text{'index'}, 0) \\ n+3 & \text{if } i = (\text{'index'}, 1) \\ n+4 & \text{if } i = (\text{'index'}, 2) \\ \text{Undefined} & \text{otherwise} \end{cases}$$

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 **Keys** to **Vals**. Strictly speaking an **Object** cannot be defined in isolation, it requires to be defined as part of a **Heap**:

$$\mathcal{H}(m) = \begin{cases} (\text{List}, n, s) & \text{if } m = n \\ 1 & \text{if } m = n+1 \\ \text{None} & \text{if } m = n+2 \\ 4 & \text{if } m = n+3 \\ (\text{Tuple}, n+4, t) & \text{if } m = n+4 \\ 0 & \text{if } m = n+5 \\ \text{Undefined} & \text{otherwise} \end{cases}$$

and

$$t(i) = \begin{cases} n+5 & \text{if } i = \text{'size' } \\ \text{Undefined} & \text{otherwise} \end{cases}$$

Semantics of Expressions

In the same manner, as Fromherz et al. (2018), we define the semantics of an $\mathbb{E}[\text{expr}]$ 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 **Iden** or a tuple **Object** \times (string \times **Val**)).

Notation: A brief note on the semantics presented below. Opposed to how the semantics were presented in Chapter 3, we present the semantics in here in raw/plain text form due to the high time they require to write properly in \LaTeX math. We apologise in advance to the readers who may have wished on polished, mathy looking equations. Please accept this cake as an apology 🍰.

```

E[expr] : Global x Heap
         → Global
         x Heap
         x (Val + (Object x (string x Val)) + Iden)

```

```

E[Name(id, ctx)](G, H) :=
  match ctx in
    case Load → if G(id) = Undefined
      -- if a variable is not in the global scope we check if it is builtin
      then if isbuiltin(id)
        then (G, H, <builtin-val>(id))
        else <Execution Halt>
      else (G, H, H(G(id)))

    -- Something will be stored in id S[Assign(...)] or variation will take care of it
    case Store → (G, H, id)

    -- The id will be deleted, S[Delete(...)] will take care of it
    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)
  in if kind(v1) ≠ Val or kind(v2) ≠ Val
    then <Execution Halt> -- error at parsing
    else let prim_op := get_prim_op(op, v1, v2)
        in prim_op(G2, H2)

E[Attribute(e, attr)](G, H) :=
  let (G1, H1, ad) := E[e](G, H)
      -- `e` must compute to a Val
      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), attr)(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))

      -- No PrimVal or <prim-callable> is subscriptable
      otherwise → <Execution Halt>

    -- 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: <prim-callable>) → call(*vals, G, H)

```


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

```
<builtin-val> : string → Val
<builtin-val>(id) :=
  match id in
    case 'int' → <prim-int-type>
    case 'list' → <prim-list-type>
    ...
```

<Execution Halt> is used in two ways in here. Either it means that we found an operation that throws an exception (which this specification 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 is the `Store` context).

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

```
get_prim_op :: Op x Val x Val → Global x Heap → Global x Heap x Val
```

```
get_prim_op(Add, t1, t2) :=
  match (type(v1), type(v2)) in
    case (Int, Int)      → <prim-+-int>(v1, v2, G, H)
    case (Float, Float) → <prim-+-float>(v1, v2, G, H)
    case (Int, Bool)    → λ(G, H) → <prim-+-int>(v1, to_int(v2), G, H)
    case (Bool, Int)    → λ(G, H) → <prim-+-int>(to_int(v1), v2, G, H)
    case (Float, a)     → if a = Bool or a = Int
                          then λ(G, H) → <prim-+-float>(v1, to_float(v2), G, H)
                          else <Execution Halt>
    case (a, Float)     → if a = Bool or a = Int
                          then λ(G, H) → <prim-+-float>(to_float(v1), v2, 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> : Val x Val x Global x Heap → Global x Heap x Val
<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 expressions, the semantics of statements do not return any kind of value, they just modify the state of the program.

```
S[stmt] :: Global x Heap → Global x Heap
```

```

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 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>
      -- e should have returned a way to find the place to remove the value

```

```

    case Addr → <Execution Halt>
    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
        -- We handle only positive cases
        if ind < lst('size') and ind ≥ 0
        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 normal expressions in the language. Thus they 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 program³ 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 implicitly or explicitly is using the `annotations` future statement.

³Well, this is not strictly true. In both, Python 3.6+ and Python 4, type annotations are stored in the special variable called `__annotations__`.

In this specification, 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 specification 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 specification 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, we extend the specification with NumPy array and discuss some of their semantics.

The following are all the things to take into account when extending our specification to handle a new type of “built-in” **Object** type:

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

$$\mathbf{Type} := \mathbf{List} \mid \mathbf{Tuple} \mid \mathbf{Module} \mid \mathbf{NdArray}$$

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

$$(\mathbf{NdArray}, n, s)$$

where $n \in \mathbb{Z}$, and

$$s(i) = \begin{cases} n + 1 & \text{if } i = \text{'shape'}$$

$$n + 5 + m + 3o & \text{if } i = (\text{'index'}, m, o) \quad \forall m \in [0, 3], o \in [0, 2]$$

$$\text{Undefined} & \text{otherwise}$$

$$t(i) = \begin{cases} n + 2 & \text{if } i = \text{'size'}$$

$$n + 3 & \text{if } i = (\text{'index'}, 0)$$

$$n + 4 & \text{if } i = (\text{'index'}, 1)$$

$$\text{Undefined} & \text{otherwise}$$

$$\mathcal{H}(m) = \begin{cases} (\mathbf{NdArray}, n, s) & \text{if } m = n$$

$$(\mathbf{Tuple}, n + 1, t) & \text{if } m = n + 1$$

$$2 & \text{if } m = n + 2$$

$$4 & \text{if } m = n + 3$$

$$3 & \text{if } m = n + 4$$

$$0 & \text{if } m = n + 5 + o \quad \forall o \in [0, 11]$$

$$\text{Undefined} & \text{otherwise}$$

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`.

$$\langle \text{prim-callable} \rangle = \dots (\text{old operations}) \mid \langle \text{prim-np-zeros} \rangle \mid \langle \text{prim-np-dot} \rangle \mid \langle \text{prim-np-abs} \rangle \mid \dots$$

Note: The values stored inside a NumPy array are considered 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 specification of the NumPy library primitives.

Nonetheless, defining formally each one of the NumPy functions above is fairly straightforward. Although, the hardest part of a formal definition 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 object⁴.

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> (G, H) :=
  let (G1, H1, (NdArray, addr, arr)) := <prim-array> (val) (G, H)
      (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, v1, v2) :=
  match (type(v1), type(v2)) in
  ... old cases
  case (NdArray, _) →
    λ (G, H) →
      let (G1, H1, ndarr) := <prim-array> (v2) (G, H)
      in <prim-+-ndarray> (v1, ndarr, G1, H1)
  case (_, NdArray) →
    λ (G, H) →
      let (G1, H1, ndarr) := <prim-array> (v1) (G, H)
      in <prim-+-ndarray> (ndarr, v2, G1, H1)
```

4. The NumPy module holding all operations is defined:

$$\text{<numpy-mod>} := (\text{Module}, -1, f_{\text{numpy}})$$

where `-1` indicates that the address is to be set only when the module is imported, and

$$f_{\text{numpy}}(s) = \begin{cases} \text{<prim-array>} & \text{if } s = (\text{'attr'}, \text{'array'}) \\ \text{<prim-dot>} & \text{if } s = (\text{'attr'}, \text{'dot'}) \\ \text{<prim-zeros>} & \text{if } s = (\text{'attr'}, \text{'zeros'}) \\ \text{<prim-ones>} & \text{if } s = (\text{'attr'}, \text{'ones'}) \\ \dots & \\ \text{Undefined} & \text{otherwise} \end{cases}$$

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

⁴The NumPy function `array` takes almost anything as an input. `array` tries to interpret its input as an array in any way it can. There is no formal definition of how the values are interpreted although its semantics can be extracted by looking at its C implementation: <https://stackoverflow.com/a/40380014>

$$\begin{aligned}
\mathbb{S}^\#[\text{Import}(name)](\mathcal{G}^\#, \mathcal{H}^\#) &:= \\
&\text{match } name \text{ in} \\
&\text{case } ("numpy",) \rightarrow \\
&\quad \text{let } (\text{Module}, arr, mod) := \langle numpy - mod \rangle \\
&\quad \quad freeaddr := get_free_addr(\mathcal{H}^\#) \\
&\quad \text{in } (\mathcal{G}^\#['numpy' \mapsto freeaddr], \mathcal{H}^\#[freeaddr \mapsto (\text{Module}, freeaddr, mod)]) \\
&\text{case } ("numpy", alias) \rightarrow \\
&\quad \text{let } (\text{Module}, arr, mod) := \langle numpy - mod \rangle \\
&\quad \quad freeaddr := get_free_addr(\mathcal{H}^\#) \\
&\quad \text{in } (\mathcal{G}^\#[alias \mapsto freeaddr], \mathcal{H}^\#[freeaddr \mapsto (\text{Module}, freeaddr, mod)])
\end{aligned}$$

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:

$$\begin{aligned}
\mathbf{Val} &:= \mathbf{PrimVal} \mid \mathbf{Object} \mid \langle \mathbf{prim-callable} \rangle \\
&\quad \mid \mathbf{Addr} \times \langle \mathbf{prim-callable} \rangle \text{ function associated to value} \\
\mathbf{PrimVal} &:= i \in \mathbb{Z} \mid j \in \text{float} \mid \text{True} \mid \text{False} \mid \text{None} \\
\mathbf{Object} &:= \mathbf{Type} \times \mathbf{Addr} \times (\mathbf{Key} \rightarrow \mathbf{Addr} \cup \{\text{Undefined}\}) \\
\mathbf{Type} &:= \text{List} \mid \text{Tuple} \mid \text{Module} \\
\mathbf{Key} &:= \mathbf{Iden} + (\text{string} \times \mathbf{Val})
\end{aligned}$$

The definition of **Val** is recursive but it is 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 (Abstract Domain) for each of the types⁵.

The simplest of all Abstract Domain is that of NoneType, NoneType[#]. **None** is the only inhabitant of NoneType, with the order:

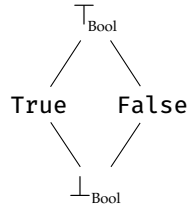
⁵A relational Abstract Domain is an Abstract Domain where the value of the variables is not assumed to be independent of each other. For more information on relational Abstract Domains look at Miné (2004). All Abstract Domains used in this work are non-relational Abstract Domains as they are simpler to understand and implement. Future work will include extending the arrange of Abstract Domains to use to some relational Abstract Domains.

$$\begin{aligned}\emptyset &\leq \emptyset \\ \{\text{None}\} &\leq \{\text{None}\} \\ \emptyset &\leq \{\text{None}\}\end{aligned}$$

The Galois connection for this order is also very simple:

$$\begin{aligned}\alpha(\{\text{None}\}) &= \text{None}^\# \\ \alpha(\emptyset) &= \perp_{\text{NoneType}}\end{aligned}\quad \text{and} \quad \begin{aligned}\gamma(\text{None}^\#) &= \{\text{None}\} \\ \gamma(\perp_{\text{NoneType}}) &= \emptyset\end{aligned}$$

A little bit more interesting is the Abstract Domain for Bool, $\text{Bool}^\#$. Bool is inhabited only by **True** and **False**. $\text{Bool}^\#$ is inhabited by four elements, namely:



Notation: Notice that in the figure above⁶, we abuse notation and write False and True for the elements $\text{False}^\#$ and $\text{True}^\#$, respectively. We have done this to keep the equations simple, but strictly speaking the elements from $\text{Bool}^\#$ are not interchangeable with the elements of Bool.

The Galois connection for this lattice is also quite simple:

$$\begin{aligned}\alpha: \mathcal{P}(\text{Bool}) &\rightarrow \text{Bool}^\# \\ \emptyset &\mapsto \perp_{\text{Bool}} \\ \{\text{True}\} &\mapsto \text{True}^\# \\ \{\text{False}\} &\mapsto \text{False}^\# \\ \{\text{True}, \text{False}\} &\mapsto \top_{\text{Bool}}\end{aligned}$$

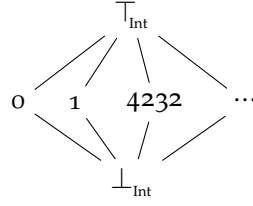
Where γ is just defined as α^{-1} given α 's bijectivity.

Notice how our previous two Abstract Domains do not require us to define widening or narrowing operators because none of them has an infinite ascending chain of values, i.e. there do not exist v_i for all $i \in \mathbb{N}$ such that $v_i \leq v_{i+1}$.

For Int and Float there are a plethora of Abstract Domains to choose from. For an in depth analysis on many of the available Abstract Domains for numbers see Miné (2004). We are going to use here the simplest Abstract Domain for number systems there is: Constant Propagation (Kildall, 1973).

Constant Propagation is very simple, in fact, the previously defined $\text{Bool}^\#$ Abstract Domain is a Constant Propagation Abstract Domain. We define Int's Abstract Domain as:

⁶This kind of diagram is called Hasse diagram. A Hasse indicates the order of the elements in a partially ordered set by means of the position of the elements in the diagram and the links between them. An element connected to other below is considered bigger or equal than the element below.



Notation: Remember that to keep things light, n^\sharp is represented as n .

In the same manner as with Bool^\sharp , we define the Galois connection as:

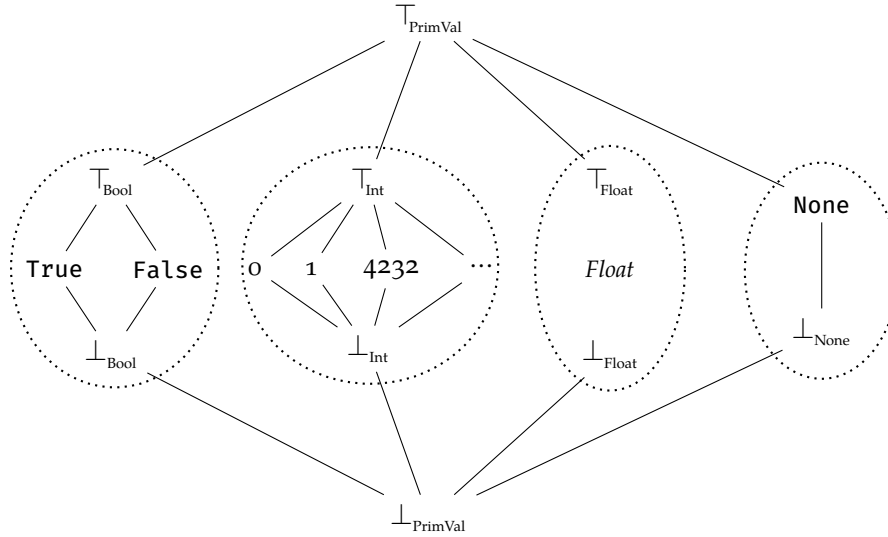
$$\begin{array}{ll}
 \alpha: \mathcal{P}(\text{Int}) \rightarrow \text{Int}^\sharp & \gamma: \text{Int}^\sharp \rightarrow \mathcal{P}(\text{Int}) \\
 \emptyset \mapsto \perp_{\text{Int}} & \perp_{\text{Int}} \mapsto \emptyset \\
 \{n\} \mapsto n & n \mapsto \{n\} \\
 \text{otherwise} \mapsto \top_{\text{Int}} & \top_{\text{Int}} \mapsto \text{Int}
 \end{array}
 \quad \text{and}$$

where

$$\text{Int}^\sharp = \text{Int} \cup \{\top_{\text{Int}}, \perp_{\text{Int}}\}$$

We define Float^\sharp just in the same way.

Now that we have an Abstract Domain for each primitive value, we can construct an Abstract Domain for **PrimVal**. The idea is simple, as shown in the figure below, we just define an Abstract Domain that groups all the Abstract Domains for primitive values together:



The formal definition is in the same lines as the others already shown. Now, consider the Galois connection:

$$\begin{aligned}
\alpha_{\text{PrimVal}} : \mathcal{P}(\text{PrimVal}) &\rightarrow \text{PrimVal}^\sharp \\
\emptyset &\mapsto \perp_{\text{PrimVal}} \\
\{n\} &\mapsto n^\sharp \\
s &\mapsto \begin{cases} \top_{\text{Int}} & \text{if } \forall m \in s : m \in \text{Int} \\ \top_{\text{Float}} & \text{if } \forall m \in s : m \in \text{Float} \\ \top_{\text{Bool}} & \text{if } \forall m \in s : m \in \text{Bool} \\ \top_{\text{PrimVal}} & \text{otherwise} \end{cases}
\end{aligned}$$

This is not the only way to define an Abstract Domain out of other Abstract Domains. In fact, there are many ways, one of which is to define an Abstract Domain where each individual Abstract Domain is extended with a new $\text{Undefined}_{\text{Type}}$ value, and they are all grouped into a tuple (Fromherz et al., 2018).

Vals Abstract Domain

Val definition

Remember **Val**'s definition:

$$\begin{aligned}
\mathbf{Val} &:= \mathbf{PrimVal} \mid \mathbf{Object} \mid \langle \mathbf{prim-callable} \rangle \\
&\quad \mid \mathbf{Addr} \times \langle \mathbf{prim-callable} \rangle \quad \text{function associated to value} \\
\mathbf{Object} &:= \mathbf{Type} \times \mathbf{Addr} \times (\mathbf{Key} \rightarrow \mathbf{Addr} \cup \{\text{Undefined}\}) \\
\mathbf{Type} &:= \mathbf{List} \mid \mathbf{Tuple} \mid \mathbf{Module} \\
\mathbf{Key} &:= \mathbf{Iden} + (\text{string} \times \mathbf{Val}) \\
\mathbf{Heap} &:= \mathbf{Addr} \rightarrow \mathbf{Val} \cup \{\text{Undefined}\}
\end{aligned}$$

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 $\mathcal{H} \in \mathbf{Heap}$.
- We say that a $(a, \mathcal{H}) \in \mathbf{Addr} \times \mathbf{Heap}$ is a **valid** value if every value defined in \mathcal{H} : $\text{vars} = \{v \in \text{Img}(\mathcal{H}) : v \neq \text{Undefined}\}$ is reachable from $\mathcal{H}(a)$, and no **Addr** inside any defined **Addr** points to Undefined.

An example of a possible value is $(0, \mathcal{H})$ where \mathcal{H} is defined as:

$$\begin{aligned}
\mathcal{H}(m) &= \begin{cases} (\mathbf{List}, 0, l) & \text{if } m = 0 \\ 3 & \text{if } m = 1 \\ \text{None} & \text{if } m = 2 \\ 4 & \text{if } m = 3 \\ (\mathbf{Tuple}, 4, t) & \text{if } m = 4 \\ 0 & \text{if } m = 5 \\ \text{Undefined} & \text{otherwise} \end{cases} \\
l(i) &= \begin{cases} 1 & \text{if } i = \text{'size'} \\ 2 & \text{if } i = (\text{'index'}, 0) \\ 3 & \text{if } i = (\text{'index'}, 1) \\ 4 & \text{if } i = (\text{'index'}, 2) \\ \text{Undefined} & \text{otherwise} \end{cases} \\
t(i) &= \begin{cases} 5 & \text{if } i = \text{'size'} \\ \text{Undefined} & \text{otherwise} \end{cases}
\end{aligned}$$

We shall remember this example from a previous subsection (see A.1.2). This heap contains the list `[None, 4, ()]` in memory.

Notice that the values $(1, \mathcal{H})$, $(2, \mathcal{H})$, ..., and $(5, \mathcal{H})$ are not considered valid as it is impossible starting from one of them to reach all other values in the Heap. We will consider only valid values for the rest of the document, which means that all equations will assume all values and states to be valid. If we were to find a non-valid value (n, \mathcal{H}) , we could define a new (n, \mathcal{H}') where \mathcal{H}' has all non-reachable values removed.

$Val^\#$ definition

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

$$\begin{aligned} \mathbf{Val}^\# &:= \mathbf{PrimVal}^\# \mid \mathbf{Object}^\# \mid \langle \mathbf{prim-callable} \rangle^\# \\ &\quad \mid \mathbf{Addr} \times \langle \mathbf{prim-callable} \rangle^\# \\ \mathbf{Object}^\# &:= \mathbf{Type} \times \mathbf{Addr} \times ((\mathbf{Key} \rightarrow \mathbf{Addr} \cup \{\mathbf{Undefined}\}) \cup \{\mathbf{ImBot}, \mathbf{ImTop}\}) \\ \mathbf{Heap}^\# &:= \mathbf{Addr} \rightarrow \mathbf{Val}^\# \cup \{\mathbf{Undefined}\} \end{aligned}$$

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

$$\begin{aligned} &(0, [0 \mapsto 3]) \quad (0, [0 \mapsto \top_{\text{Int}}]) \quad (0, [0 \mapsto \perp_{\text{Bool}}]) \quad (0, [0 \mapsto \top_{\text{Val}}]) \\ &(0, [\\ &\quad 0 \mapsto (\mathbf{List}, 0, ['\text{size}' \mapsto 1, ('index', 0) \mapsto 2, ('index', 2) \mapsto 0, ('index', 6) \mapsto 3]) \\ &\quad 1 \mapsto \top_{\text{Int}} \\ &\quad 2 \mapsto 21.3 \\ &\quad 3 \mapsto (\mathbf{Tuple}, 3, \mathbf{ImBot}) \\ &]) \end{aligned}$$

Notation: Once more, we are abusing notation for convinience. $[]$ represents a function which codomain is only $\mathbf{Undefined}$, and $[0 \mapsto 5, 1 \mapsto \mathbf{None}]$ represents a function which output to the in puts 0 and 1 is 5 and \mathbf{None} , respectively.

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

We have defined the first ingredient of the $\mathbf{Val}^\#$ Abstract Domain. The steps left are:

- Define abstraction α and concretisation γ functions,
- define an order relation,
- join $(\sqcup_{\mathbf{Val}^\#})$ and merge $(\sqcap_{\mathbf{Val}^\#})$ operations, and
- a Galois connection.

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

We will start by defining the *join* operation and the rationale behind its inner workings. 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])
-- checking all other cases TopVal = TopVal, <prim-_-> = <prim-_->, ...
else if on = om
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_empty(a,b) := Undefined
```

```
H_empty: Heap#
```

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

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

- \perp_{Val} must be the lowest value in the order, therefore any value joining with it should be the same value ($\perp_{\text{Val}} \sqcup n = n$).
- \top_{Val} is the biggest value in the order, therefore any value joining with it should give back \top_{Val} ($\top_{\text{Val}} \sqcup n = \top_{\text{Val}}$).
- 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 $\sqcup_{\text{PrimVal}^\#}$.
- Joining **Object**[#]s should take into account the recursive nature of the definition of **Val**[#]s and **Object**[#]s.

Without further ado, the definition of $\sqcup_{\text{Val}^\#}$ (*joinVal*):

```
HeapCon# = Addr → Val# + Undefined + InConstruction
```

```
-- we assume that (n,H1#) and (m,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_right -- n or m has already 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 another 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)→ad], 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)→ad], 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 fm, 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) × Heap#) × ((Key → Addr + Undefined) × Heap#)
  × (Addr × Addr → Addr + Undefined) × HeapCon#
  → (Key → Addr + Undefined) × (Addr × Addr → Addr + Undefined) × 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) × (Addr × Addr → Addr + Undefined) × HeapCon#)
    × Key
    → ((Key → Addr + Undefined) × (Addr × Addr → Addr + Undefined) × HeapCon#)
  helper((fnnew, joined, H_new#), key) :=
    if fm(key) = Undefined
    then let (joined', H_new'#) :=
        makeallreachable_top_left((fn(key), H1#), joined, H_new#)
        ad := freeadd(H_new'#)
        in (fnnew[key→ad], joined', H_new'#[ad→Top_Val])
    else if fn(key) = Undefined
    then let (joined', H_new'#) :=
        makeallreachable_top_right((fm(key), H2#), joined, H_new#)
        ad := freeadd(H_new'#)
        in (fnnew[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 (fnnew[key→n'], joined', H_new'#)
    else if val1 is PrimVal# and val2 is PrimVal#
    then (fnnew[key→ad], joined[(adn,adm)→ad], H_new#[ad→val1 UPrimVal# val2])
    else if on = Bot_Val
    then (fnnew[key→ad], joined[(adn,adm)→ad], H_new#[ad→InConstruction])
    else if val2 = Bot_Val
    then (fnnew[key→ad], joined[(adn,adm)→ad], H_new#[ad→InConstruction])
    -- checking all other cases TopVal = TopVal, <prim-_-> = <prim-_->, ...
    else if on = val2
    then (fnnew[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_empty, joined, H_new#), PreIm(fn) U PreIm(fm))
```

Notice that it is in `joinVal` ($\sqcup_{\text{Val}^\#}$) where the whole magic of this Abstract Domain lies. `joinVal` is meant to walk through both graphs simultaneously, find the similarities, implode the differences between the graphs and preserve all parts that coincide.

The definition of `joinVal` requires the help of the function `joinObjectFn` which joins the values to which an **Object[#]** points (the function $(\text{Key} \rightarrow \text{Addr} \cup \{\text{Undefined}\})$).

`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 $\mathcal{H}^\#$ but $\text{Addr} \rightarrow \text{Val}^\# \cup \{\text{Undefined}, \text{InConstruction}\}$, i.e. the output of `Hnew#` is the same of $\mathcal{H}^\#$ but it has an additional value called `InConstruction`. 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 any reference to some `InConstruction` value left. All `InConstruction` values left after returning from the whole walk will only be those which appeared when joining some node to a `ImBot` or `Bot_Val`. Because we expect $\text{ImBot} \sqcup fn = fn$, we must copy all values from the joining heaps into the new heap.

Val[#] Order definition

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

We define \leq as $a \leq_{\text{Val}^\#} b \iff b = a \sqcup_{\text{Val}^\#} b$. This definition follows from the definition of *join*: given any two values $a, b \in \text{lattice}$, 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 result of *joining* a with b , otherwise there would be a value smaller than b that is also bigger or equal than b . What we are trying to argue in here is that we can define a lattice by either defining an order relation or a proper *join* operation for it. We leave the proof that \leq defines an order for valid values $((a, \mathcal{H}^\#) \in \text{Val}^\# \times \text{Heap}^\#)$ for future work.

Other definitions

Our goal is not to present in detail all functions and operations necessary to define the Abstract Domain for **Val[#]**s. That would require three times the space already used in this document. What we 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 of different types. *Joining* values with differing types gives us \top values, while *merging* gives us \perp values.

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:

$$\begin{aligned}\mathbf{Global}^\# &:= \mathbf{Iden} \rightarrow \mathbf{Addr} \cup \{\mathbf{Undefined}\} \\ \mathbf{Heap}^\# &:= \mathbf{Addr} \rightarrow \mathbf{Val}^\# \cup \{\mathbf{Undefined}\}\end{aligned}$$

Notice that $\mathbf{Global}^\#$ is a function that takes an identifier and outputs an address to where the variable is stored, which is just the same thing to what the function ($\mathbf{Key} \rightarrow \mathbf{Addr} \cup \{\mathbf{Undefined}\}$) inside $\mathbf{Object}^\#$ does. In fact, the function inside $\mathbf{Object}^\#$ has a bigger pre-image than that of $\mathbf{Global}^\#$ (\mathbf{Key} is defined as $\mathbf{Iden} + (\text{string} \times \mathbf{Val})$). This means that we have no need to define a *join* operation from scratch for the State Abstract Domain. We can just borrow/repurpose the already defined function `joinObjectFun`, defined for $\mathbf{Object}^\#$ s to $\mathbf{Global}^\#$ s!

```
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, we present some examples of the abstract semantics. Notice that we do not need the abstraction and concretisation functions as the State and State Abstract Domains are quite similar. It is left for future work to prove 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# + (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(v2) ≠ 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, v1, v2)
  in prim_op#(G2#, H2#)

get_prim_op : Op x Val# x Val# → Global x Heap → Global x Heap x Val#

get_prim_op#(Add, v1, v2) :=
  match (type(v1), type(v2)) in
  case (Int, Int) → λ(G#, H#) → <prim-+-int>#(v1, v2, G#, H#)
  case (Float, Float) → λ(G#, H#) → <prim-+-float>#(v1, v2, G#, H#)
  case (Int, Bool) → λ(G#, H#) → <prim-+-int>#(v1, Int(v2), G#, H#)
  case (Bool, Int) → λ(G#, H#) → <prim-+-int>#(Int(v1), v2, G#, H#)
  case (Float, a) → if a = Bool or a = Int
    then λ(G#, H#) → <prim-+-float>#(v1, Float(v2), G#, H#)
    else <prim-ret-top>#
  case (a, Float) → if a = Bool or a = Int
    then λ(G#, H#) → <prim-+-float>#(Float(v1), v2, G#, H#)
    else <prim-ret-top>#
  case otherwise → <prim-op-top>#

<prim-op-top># : Global# x Heap#
<prim-op-top>#(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 is 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 concrete semantics must be rewritten as function that operates in the Abstract Domain. For example, consider the function `<prim-+-int>`, which 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)])

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(...)]
```

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.
- Cannon, B. (2005). *Localized type inference of atomic types in python* (PhD Thesis, Citeseer).
- 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-detyped: Reexporting tensorflow haskell with dependent typed functions. <https://github.com/helq/tensorflow-haskell-detyped>. GitHub.
- Eaton, F. (2006). Statically typed linear algebra in Haskell. In *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell* (pp. 120–121). ACM.
- Eilers, M., & Müller, P. (2018, July 14). Nagini: A static verifier for python. In *Computer aided verification* (pp. 596–603). Lecture Notes in Computer Science. International conference on computer aided verification. doi:10.1007/978-3-319-96145-3_33
- 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/>
- Gunter, C. A. (1992). *Semantics of programming languages: Structures and techniques*. MIT press.
- Guth, D. (2013). A formal semantics of Python 3.3.
- Hassan, M., Urban, C., Eilers, M., & Müller, P. (2018). MaxSMT-Based Type Inference for Python 3. In *Computer Aided Verification* (pp. 12–19). Lecture Notes in Computer Science. doi:10.1007/978-3-319-96142-2_2
- Hayes, B. (2003). Computing science: A lucid interval. *American Scientist*, 91(6), 484–488.
- Hejlsberg, A. (2012). Introducing typescript. *Microsoft Channel*, 9.
- 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>
- Kildall, G. A. (1973). A unified approach to global program optimization. In *Proceedings of the 1st annual acm sigact-sigplan symposium on principles of programming languages* (pp. 194–206). ACM.
- 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).
- Mitchell, J. C. (1996). *Foundations for programming languages*. MIT press Cambridge.
- 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>
- Müller, P., Schwerhoff, M., & Summers, A. J. (2016). Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann & K. R. M. Leino (Eds.), *Verification, model checking, and abstract interpretation (vmcai)* (Vol. 9583, pp. 41–62). LNCS. Springer-Verlag.
- Nielson, F., Nielson, H. R., & Hankin, C. (2005). *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
- Norwitz, N. (n.d.). Pychecker. Retrieved from <http://pychecker.sourceforge.net>
- 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).
- Slepek, 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
- Thenault, S. et al. (2006). Pylint – code analysis for python. [Online; accessed May 21 2018]. Retrieved from <https://pylint.org/>
- Trojahn, 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>
- Van Rossum, G. et al. (2007). Python programming language. In *Usenix annual technical conference* (Vol. 41, p. 36).