Program Analysis: Towards the Analysis of CPython Bytecode

André Theuma

Supervisor: Dr. Neville Grech

May, 2022

Submitted in partial fulfilment of the requirements for the degree of B.Sc. Computer Science.

L-Università ta' Malta
Faculty of Information & Communication Technology

Acknowledgements

These are the acknowledgements. Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Abstract

Program analysis methods offer static compile-time techniques to predict approximations to a set of values or dynamic behaviours which arise during a program's run-time. These methods generate useful observations and characteristics about the underlying program, in an automated way.
PATH (Python Analysis Tooling Helper) is a static analysis tool created in this project, which generates a standardized Intermediary Representation for given functions, allowing analysis metrics to be generated from the facts produced by the tool. The goal of this project was to create a framework that generates facts from a function, in addition to an IR that is amenable for further analysis. The framework created should simplify the engineering complexity of fact analysis for future use. PATH would disassemble CPython bytecode into a more straightforward representation, making any further possible analyses a simpler task, as analysis can be conduced on the generated IR.
The final findings of the project indicate that performing analysis on the IR generated by PATH is indeed a simpler task than generating facts manually and conducting block analysis without such a framework. These results are satisfactory and hold up to the aims of this project.

Contents

List of Figures

List of Tables

List of Abbreviations

AIL Abstract Interpreter Loop
PATH Python Analysis Tooling Helper
IR Intermediary Representation
MD5 Message-Digest 5 Algorithm
DFA Data Flow Analysis
CFA Control Flow Analysis
CFG Control FLow Graph
III Iterative Instruction Inspection
I/O Inputs & Outputs

Chapter 1

Introduction

1.1 Motivation

As an increasing number of people are becoming reliant on complex software systems, the lack of security and analysis tooling frameworks is unacceptable in this day and age. Such frameworks provide vital insight into software systems; allowing further development and refinement of said systems. Core vulnerabilities are exposed, and optimizations are possible. Tools such as Pylint Logilab (2022), and Bandit Developers (2022) already exist in the industry.

1.2 Preliminary Overview

1.2.1 Python & CPython

(andre:) Reworked

Python is a high-level, object-oriented scripting language Lutz (2001), suited for a wide range of development domains; from text processing Bonta and Janardhan (2019) to machine learning Goldsborough (2016) to game development Sweigart (2012). The language's wide adoption (TIOBE's Language of the Year: 2007, 2010, 2018, 2020, 2021; Index (2022)) may be attributed to the fact that it is based on the English language Saabith et al. (2019), making it easy to learn; aiding in the production of relatively complex programs. It is used extensively for rapid prototyping and the development of fully-fledged real-world applications.

The predominant and most comprehensive implementation of Python is known as CPython van Rossum; a bytecode interpreter for Python, written in C.

1.2.2 Program Analysis

(andre:) Reworked

Complex programs imply complex behaviours. Such behaviours have to be analysed, as they might highlight potential vulnerabilities, and possibly indicate where optimisations can be carried out in the program. This area of interest is known as Program Analysis. Program Analysis provides answers to the following questions;

- Can the provided code be optimized?
- Is the provided code free of errors?
- How does data flow through the program & in what order do instructions get executed (Control-Flow)?

(andre:) Reworked

Naturally, as an increasing amount of modern-day systems and frameworks are developed in Python, the need for conducting program analysis on these systems is ever-growing. There are two main approaches to program analysis; Dynamic Program Analysis & Static Program Analysis. Dynamic analysis is the testing and evaluation of an application during runtime, whilst static analysis is the testing and evaluation of an application by examining the code, producing facts and deducing possible errors in the program from the facts produced; without code execution Intel (2013). Since all (significant) properties of the behaviour of programs written in today's programming languages are mathematically undecidable Rice (1953), one must involve approximation for an accurate analysis of programs. This kind of analysis cannot be carried out by a Dynamic analysis as carrying out a runtime analysis only reveals errors, but does not show the absence of errors Møller and Schwartzbach (2012); being the primary motivation behind Static analysis. With the right kind of approximations, Static analysis provides guarantees regarding the properties of all the possible execution paths a program can take, giving it a clear advantage over Dynamic analysis; thus will be the main topic of interest in this paper.

1.3 Proposed Solution

PATH provides general metrics for functions; known as facts, along with a standardized IR (Intermediary Representation) for external analysis. PATH also creates a Control Flow Graph for flow analysis.

1.3.1 Aims & Objectives

(andre:) Reworked

The aim of this dissertation is the production of an easily implementable analytical tool (PATH) for existing software systems. The tool is to be used on Python V3.10 van Rossum systems which are interpreted with the CPython van Rossum interpreter. It needs to scale up to larger systems and still provide accurate metrics. The analytical tool must also generate a standardized IR of the functions present in the system that are analysed. The standardized generated IR has to summarize the block analysis done by PATH.

The creation of this tool is vital as there is a lack of security and analysis tooling, for what is the world's most used programming language Index (2022); giving developers increased freedom of choice when having to choose an analysis framework for their projects.

1.4 Document Structure

This paper is composed of five chapters; broken down in the following manner:

Chapter 1 contains the introductory content, along with a brief overview of the technologies and ideas explored further along in this paper. The introduction also gives a run-down of the main objectives met in this project.

Chapter 2 gives an in-depth literature review of the content briefly touched upon in the introductory chapter [Chapter 1]. The literature review consists of (but is not limited to) the following works; Python, the uses of Python, Python's bytecode & CPython's *ceval.c* interpreter, and Static Analysis Tooling.

Chapter 3 delves into both the methodology and implementation of the disassembler (PATH). The design of choice is further discussed together with the reasoning behind such a design.

Chapter 4 evaluates the results produced by PATH and answers specific research questions, ensuring that the Analytical toolkit works as intended. A couple of case studies are included, testing the scalability and ease-of-use of PATH.

Chapter 5 presents the conclusions of the project and any suggestions for any possible further work.

1.5 Contributions

(andre:) new

A.T carried out the reverse engineering of the individual CPython bytecodes; observing their effect on the value stack. These operations are concisely noted in the Opcode Table.

A.T automated the production of facts from Python functions, expediting different analyses of functions.

A.T standardized the generation of an IR; exposing to the user for better understanding of function behaviours. Alongside IR generation, control flow with functions is indicated by a Control-Flow graph.

Chapter 2

Background & Literature Overview

2.1 Python

2.1.1 History

Python being a platform-independent (Srinath, 2017), high-level programming language (Van Rossum and Drake Jr, 2020, pp.2–4), makes the development of complex software systems a relatively non-trivial task in comparison to the production complexity that comes along with other comparable programming languages, such as C (Summerfield, 2007).

The language was developed to be a successor of the ABC programming language (Geurts et al., 1990) and initially released as Python V0.9.0 in 1991. Similarly to ABC, Python has an English-esque syntax but differs in its application domain; Python being a tool intended for use in a more professional environment, whilst ABC is geared towards non-expert computer users (van Rossum and de Boer, 1991, pp.285–288).

2.1.2 Features & Philosophy

The simplicity of Python enables it to be an extremely popular language to use in the industry of software development (Index, 2022). It was designed to be highly readable, thus removing the 'boilerplate' style used in the more traditional languages, such as Pascal. Python uses indentation for block delimiters, (off-side rule (van Rossum, pp.4–5)) which is unusual among other popular programming languages (van Rossum, pp.2–3); new blocks increase in indentation, whilst a decrease in indentation signifies the end of a block. It supports a dynamic type system, enabling the compiler to run faster and the CPython interpreter to dynamically load new code. Dynamic type systems such as Python offer more

flexibility; allowing for simpler language syntax, which in turn leads to a smaller source code size (Thomas, 2013). Although dynamically typed, Python is also strongly-typed; disallowing operations which are not well-defined. Objects are typed albeit variable names are untyped. (andre:) Reworked

One of Python's most attractive features is that it offers the freedom to allow the developer to use multiple programming paradigms (Van Rossum et al., 2007); appealing to a wider audience. Python also includes a cycle-detecting garbage collector (Van Rossum et al., 2007), freeing up objects as soon as they become unreachable (Van Rossum and Drake Jr, 1994, pp.9–10). Objects are not explicitly freed up as the collector requires a significant processing overhead (Zorn, 1990, pp.27-30), and re-allocating memory to objects every time an object is required is resource consuming. Python has module support in its design philosophy, formulating a highly extensible language. Modules can be written in C/C++ (Srinath, 2017) and imported as libraries in any Python project; highlighting the extensibility of the language. There are plenty [1] of readily available third-party libraries suited for many tasks, ranging from Web Development (Forcier et al., 2008) to complex Machine Learning frameworks (Pedregosa et al., 2011), further increasing ease-of-use, and supporting the quick and simple development philosophy of Python.

2.1.3 Implementations

(andre:) Reworked

There are several environments which support the Python language, known as implementations. The default, most feature comprehensive Python implementation is CPython (Van Rossum, 1995); written and maintained by Guido van Rossum. Other popular re-implementations include PyPy (Bolz et al., 2009), Jython (Juneau et al., 2010) and IronPython (Mueller, 2010). This paper will focus on the CPython implementation of the Python language and will not cover any of the other alternate implementations.

2.2 CPython

2.2.1 Overview

(andre:) Reworked

CPython is the predominant implementation of the Python language written in C. It has a thin compilation layer from source code to CPython bytecode (See 2.2); simplifying

[1] Over 329,000 packages as of September 2021 (Van Rossum et al., 2007)

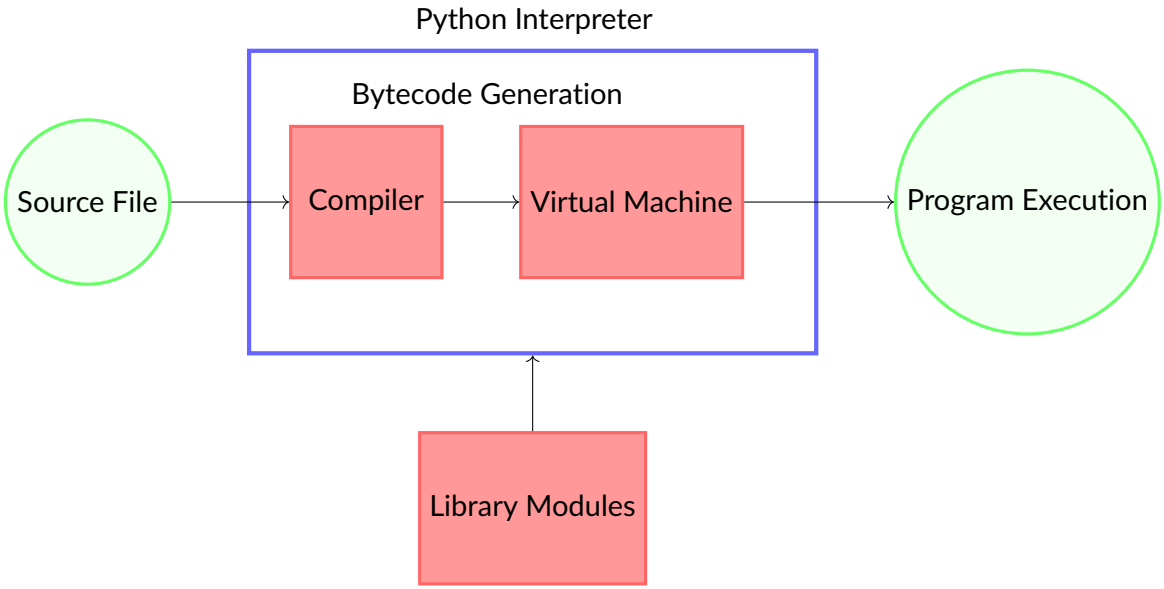

Figure 2.1: Python Code Execution

the design of the interpreter. Unlike the typically structured program representations, bytecode is easier to parse and has a standardized notation.

CPython works transparently, via the PVM(Python Virtual Machine 2.3); an interpreter loop is run [2] and there is no direct translation between the Python code to C (Aycock, 1998, pp.1–2). The PVM is a stack machine whereby PVM instructions retrieve their arguments from the stack, just to be placed back onto the stack after execution of the instruction. The Python compiler generates PVM code[3] for the python VM to execute. The CPython interpreter resembles a classic interpreter with a straightforward algorithm (Aycock, 1998, pp.2–4):

1. Firstly, the opcode of a VM instruction is fetched, along with any necessary arguments.
2. Secondly, the instruction is then executed.
3. Finally, steps 1-2 are repeated till no more opcodes can be fetched. This is done by raising an exception when an invalid (empty) opcode is found.

This simple algorithm is known as the CPython Evaluation Loop [2]. The evaluation loop [2] is formulated in the following manner[4]:

[2] *ceval.c*

[3] PVM code is also known as a *.pyc* file.

[4] Actual code differs from what is presented. The source code has been edited as to be more readable and concise.

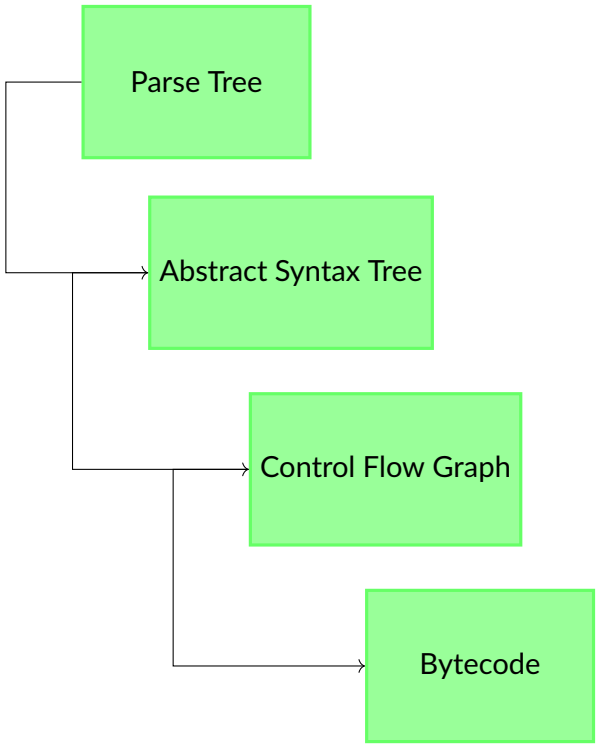

Figure 2.2: Compiler Innards

Evaluation is computed frame by frame (See **Frames**), with what is essentially a (very long) switch statement; reading every opcode and delegating accordingly.

2.2.2 Stacks

A stack data structure is a dynamic structure that operates with a LIFO policy (Cormen et al., 2009). Since CPython does not directly interact with the hardware for compilation, it makes both the call stack and stack frames rely on the PVM. In CPython, there is one main stack that the PVM requires for proper functionality; the call stack. The other two stacks (Value Stack and Call Stack) are essential for the proper computation of any variables that there are in the frame (See **Frames**). Most instructions manipulate the value stack and the call stack (Bennett, 2018). The nature of CPython stacks is visualised in Figure 2.5

Call Stack

(andre:) Reworked

The call stack contains call-frames[5] (See **Frames**). This is the main structure of the running program. A function call results in a pushed frame onto the call stack, whilst a

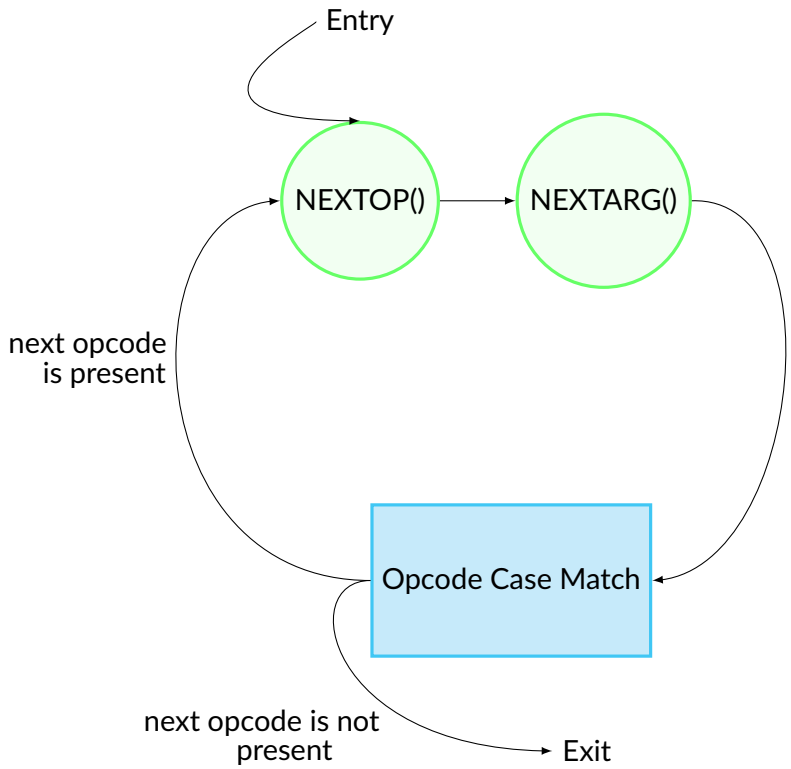

Figure 2.3: VM Innards

```
      for(**indefinite condition**){
        oparg=null;
        opcode = NEXTOP();
        if(ARGUMENT_PRESENT(opcode)){
          oparg = NEXTARG();
        }
        switch(opcode){
          case **opcode_name**:
            manipulate stack & set variables accordingly
          ...
          ...
          ...
          default:
            raise error
        }
      }
```

Listing 2.1: Evaluation Loop

return call results in a pop of the function frame off of the stack (Conrad, 2010). A visual representation of a call stack is shown in Figure 2.4. In this figure, a sample script can be seen run, step by step, showing the frames being pushed onto the call stack and popped from the call stack. The first frame pushed onto the frame stack is inevitably called the `__main__` call frame.

Value Stack

(andre:) Reworked

This stack is also known as the evaluation stack. It is where the manipulation of the object happens when evaluating object-manipulating opcodes. A value stack is found in a call-frame, implying bijectivity. Any manipulations performed on this stack (unless they are namespace related) are independent of other stacks and do not have the permissions to push values on other value stacks.

Block Stack

The block stack keeps track of different types of control structures, such as; loops, try/except blocks, and with blocks. These structures push entries onto the block stack, which are popped whenever exiting the said structure. The block stack allows the interpreter to keep track of active blocks at any moment

```
1           def foo():
2             print("Hello")
3
4           def intermediary():
5             foo()
6
7           def start():
8             intermediary()
9
10          start()
11
```

Listing 2.2: Python script

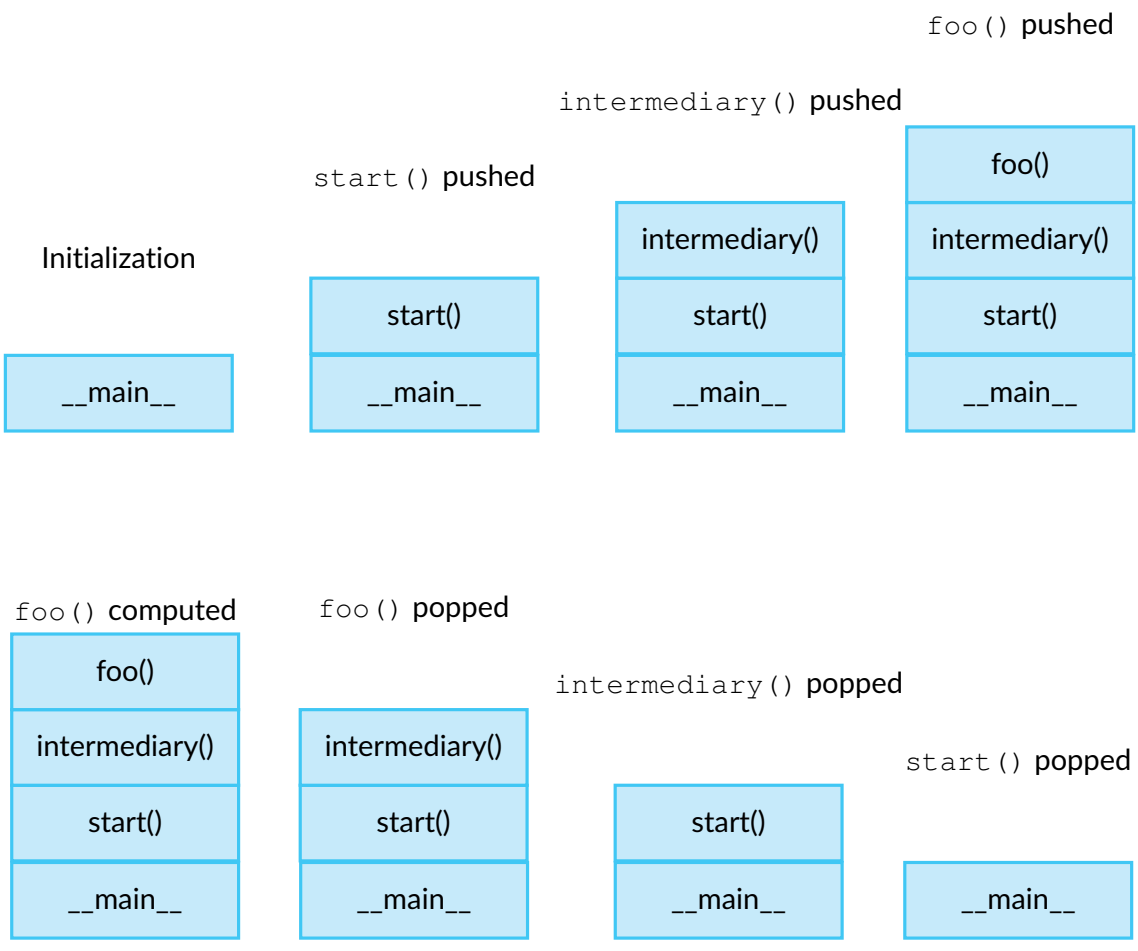

Figure 2.4: Simulation of Call Stack running Listing **??**

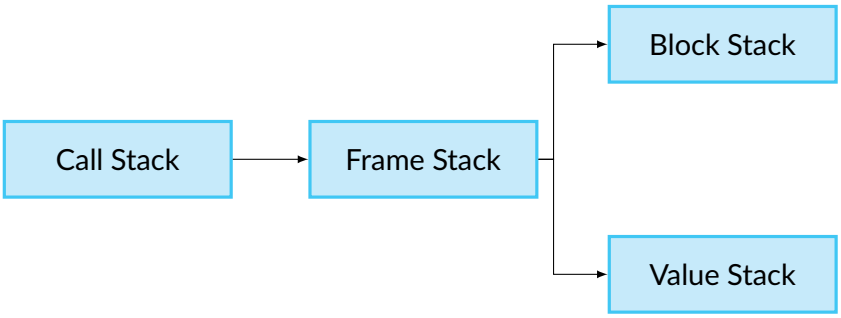

Figure 2.5: Overview of CPython stacks

2.2.3 Frames

(andre:) Maybe change???

A frame[5], is an object which represents a current function call (subprogram call); more formally referred to as a code object. It is an internal type containing administrative information useful for debugging and is used internally by the interpreter (Van Rossum and Drake Jr, 1994, pp.18–19). Frame objects are tightly coupled with the three main stacks (See **Stacks**) and every frame is linked with another frame. Every frame object has two frame-specific stacks; value stack (See **Value Stack**) and the block stack (See **Block Stack**). Frames are born from function calls, and die when that function is returned.

Frame Attributes

Along with the properties mentioned above, a frame object would have the following attributes [6]:

- `f_back`: this is[7] the previous stack frame object (return address).
- `f_code`: this is[7] the current code object (See **Code Objects**) being executed, in the current frame.
- `f_builtin`: this is[7] the builtin symbol table.
- `f_globals`: this is[7] the dictionary used to look up global variables.
- `f_locals`: this is[7] the symbol table used to look up local variables.

[5] Also called a call-frame.
[6] Retrieved from declaration found in *./Include/frameobject.h*
[7] In this context; "...this is..." or "...this holds..." does not mean the actual value is being stored in the variable, but a pointer to the address of the value is being stored.

- `f_valuestack`: this is a pointer, which points to the address after the last local variable.
- `f_valuestack`: this holds[7] the value of the top of the value stack.
- `f_lineno`: this gives the line number of the frame.
- `f_lasti`: this gives the bytecode instruction offset of the last instruction called.
- `f_blockstack`: contains the block state, and block relations.
- `f_localsplus`: is a dynamic structure that holds any values in the value stack, for evaluation purposes.

Frame Stack

(andre:) Reworked

The stack frame is a collection of all the current frames in a call-stack-like data structure (See **Call Stack**). A frame is pushed onto the stack frame for every function call (as typically, every function has a unique frame) as shown in Figure 2.6.

The stack frame contains a frame pointer which is another register that is set to the current stack frame. Frame pointers resolve the issue created when operations (pushes or pops) are computed on the stack hence changing the stack pointer, invalidating any hard-coded offset addresses that are computed statically, before run-time Bacon (2011). With frame pointers, references to the local variables are offsets from the frame pointer, not the stack pointer.

2.2.4 Code Objects

A code object is a low-level detail of the CPython implementation. When parsing Python code, compilation creates a code object for processing on the PVM. A code object contains a list of instructions directly interacting with the CPython VM; hence coined a low-level detail. Typically, code objects are of the type `PyCodeObject`[8] and each section of the code object represents a chunk of executable code that has not been bound to a function (Foundation, 2022b). The structure of the type of these code objects [8] change throughout different CPython versions, thus there is no set composition (Foundation, 2022b). For reference, the source code in Listing 2.2.4 produces the code object displayed in Listing 2.2.4, following the standard convention shown in Listing 2.2.4.

[8]The type is the C structure of the object which is used to describe a code object.

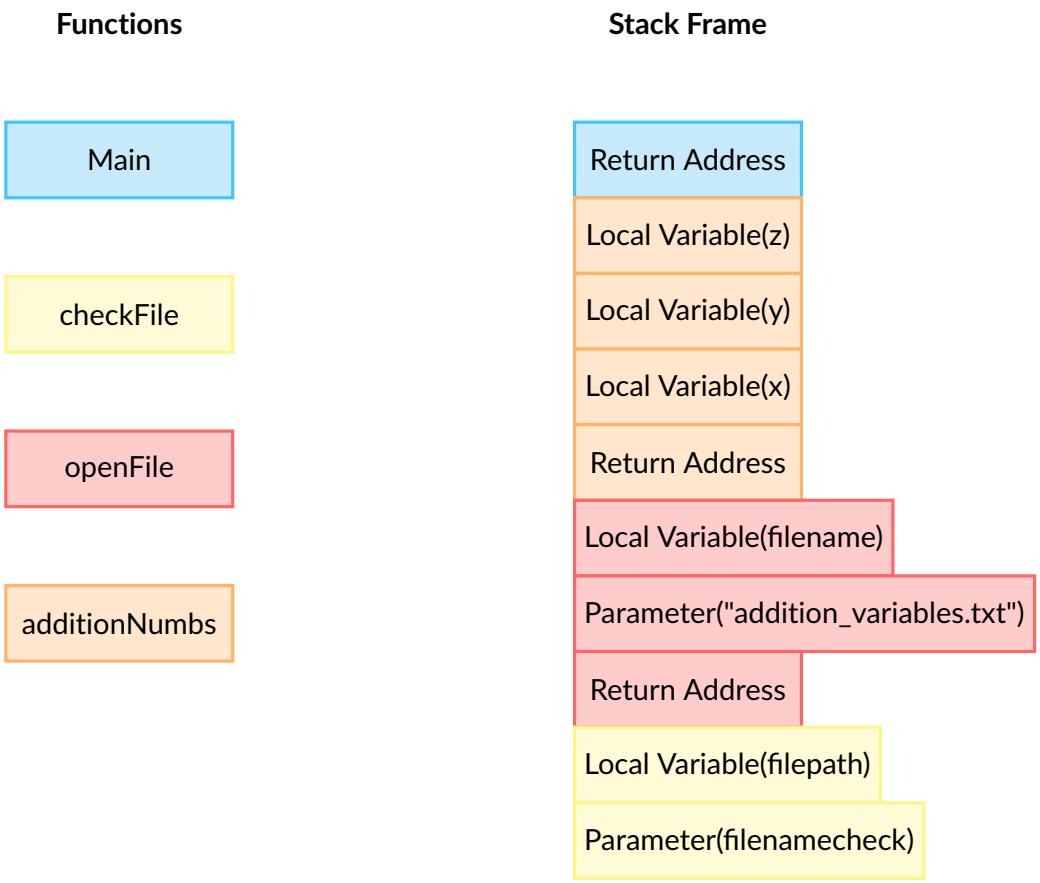

Figure 2.6: Stack frames example

```
    <code object addition_numbers at 0x1047f10b0, file "
filepath", line 3>
```

Listing 2.3: Code object of 2.2.4

```
    code object <functionName> at <address>, file <path>, line<
firstLineNo.>
```

Listing 2.4: Code object standard convention

Disassembler

(andre:) Reworked

Code objects are expanded by using the `dis` module in Python. This module contains several analyses functions which; all of which directly convert the input code object into

the desired output (Foundation, 2022a). The function that is of a particular interest in this paper is the `dis.dis` function which disassembles a code object into its respective bytecodes, alongside other relevant information, as seen in Figure 2.2.4. When applying the analysis function `dis.dis`, the disassembled code object takes the following format for every instruction:

```
<lineNumber><label><instructionOffset><opname><opargs><var>
```

Listing 2.5: Dissasembled instruction convention

It is interesting to note that the value for `opargs` is computed in little-endian order. Typically, as is shown in ... ,the arguments associated with the instructions are used for specific stack manipulations.

```
1  1          import dis
2  2
3  3          def addition_numbers(x,y):
4  4              z=x+y
5  5              return z
6  6
7  7          dis.dis(addition_numbers)
8
```

Listing 2.6: Python source code

```
4           0 LOAD_FAST          0(x)
            2 LOAD_FAST          1(y)
            4 BINARY_ADD
            6 STORE_FAST         2(z)

5           8 LOAD_FAST          2(z)
           10 RETURN_VALUE
```

Listing 2.7: Disassembly of Listing 2.2.4

Bytecode

(andre:) Reworked

Bytecode is a form of portable code executed on a virtual machine. The introduction of the concept of bytecode came about when a generalized way of interpreting complex

programming languages was required to simplify information structures, characterizing them in their essentials (Landin, 1964). This ideology gave birth to portable code[9], which is a generalized form of code, that can cross-compile, considering the Virtual Machine that interpreted the p-code was compatible with the native machine architecture.

In CPython V3.10, there are over 100 low-level bytecode instructions (Foundation, 2022a). The classification of bytecode operations is defined below:

- Unary instructions.
- Binary instructions.
- Inplace instructions.
- Coroutine instructions.
- Argument instructions.
- Miscellaneous instructions.

All these instructions are discussed with much further detail in the ... section

2.2.5 Execution of Code Objects

(andre:) Reworked The evaluation stage firstly makes use of the public API `PyEval_EvalCode()` (Rossum et al., 2022, lines 716–724), which is used for evaluating a code object created at the end of the compilation stages (Figure 2.2). This API constructs an execution frame from the top of the stack by calling `_PyEval_EvalCodeWithName()`. The first execution frame constructed must conform to these requirements:

- The resolution of keyword[10] and positional arguments[11].
- The resolution *args[12] and **kwargs[12] in function definitions.
- The addition of arguments as local variables to the scope (A scope is the membership of a variable to a region).
- The creation of Co-routines and Generators (Tismer, 2000, pp.2–3).

[9] Portable code, p-code and bytecode can be used interchangeably.

[10] A keyword argument is a value that, when passed into a function, is identifiable by a specific parameter name, such as variable assignment.

[11] A positional argument is a value that is passed into a function based on the order in which the parameters were listed during the function definition.

[12] *args and **kwargs allow multiple arguments to be passed into a function via the unpacking (*) operator.

Code execution in CPython is the evaluation and interpretation of code object. Below, we delve into a more detailed description of frame objects; their creation, and execution (Shaw, 2022).

Thread State Construction

Prior to execution, the frame would need to be referenced from a thread. The interpreter allows for many threads to run at any given moment. The thread structure that is created is called `PyThreadState`.

Frame Construction

Upon constructing the frame, the following arguments are required:

- `_co`: A `PyCodeObject` (code object).
- `globals`: A dictionary relating global variable names with their values.
- `locals`: A dictionary relating local variable names with their values.

It is important to note that there are other arguments that might be used but do not form part of the basic API, thus will not be included.

Keyword & Positional Argument Handling

If the function definition contains a multi-argument keyword argument [12], then a new keyword argument dictionary [13] must be created in the form of a `PyDictObject` Similarly, if any positional arguments[12] are found they are set as local variables.

The dictionary[13] that was created is now filled with the remaining keyword arguments which do not resolve themselves to positional arguments. This resolution comes after all the other arguments have been unpacked. In addition, missing positional arguments [14] are added to the `*args`[12] tuple. The same process is followed for the keyword arguments; values are added to the `**kwargs` dictionary and not a tuple.

Final Stage

Any closure names are added to the code object's list of free variable names, and finally, generators and coroutines are handled in a new frame. In this case, the frame is not pre-evaluated but it is evaluated only when the generator/coroutine method is called to execute its target.

[13] Denoted as a `kwdict` dictionary.

[14] Positional arguments that are provided to a function call, but are not in the list of positional arguments.

2.2.6 Execution of Frame Objects

The local and global variables are added to the frame preceding frame evaluation, which is handled by `_PyEval_EvalFrameDefault()`.

`_PyEval_EvalFrameDefault()` is the central function which is found in the main execution loop. Anything that is executed in CPython goes through this function and forms a vital part of interpretation.

2.3 Analysis

(andre:) Add hyppref to sections

Program analysis is constituted of four main approaches; Control Flow analysis, Abstract Interpretation, Type and Effect Systems, and finally Data Flow Analysis (Nielson et al., 2004, pp.1–2). Typically, these approaches are practised in conjunction with each other to provide the most accurate approximate answers. Program analysis techniques should be semantics-based and not semantics directed; the latter is the process by which the information obtained from the analysis conducted is proved to be safe concerning the semantics of the programming language, whilst the former is the process by which the structure of the analysis conducted reflects the structure of the semantics of the language; a process which is not recommended (Nielson et al., 2004, pp.2–3). These approaches also have two main methodologies driving them; Statically analysing a program or Dynamically analysing a program.

Program analysis is conducted before program input, rendering any analysis undecidable. In complexity theory, this is known as Rice's Theorem (Rice, 1953). The analysis would need to compute results that are valid for all possible inputs into the program. Seeing as such a statement is near impossible to back up, the aforesaid analysis approximates; producing a *safe* answer (Andersen, 1994, pp.9–11). *Safe* answers are decidedly *safe* based upon the aim of the analysis and the information provided to the analysis. A result which might be considered *safe* in a certain analysis, may not be in other analyses.

(andre:) include info about basic blocks

(andre:) maybe include static+dynamic analysis as well?

2.3.1 Control Flow Analysis

Control Flow analysis (*Constraint Based analysis*) is the act of determining information about what elementary blocks lead to other blocks, whereby seeing the flow of program control. More formally, such an analysis for each function application gives us which

functions may be applied. Control Flow analysis makes use of the constraint system. The essence of this method is to extract several inclusions out of a program. This system creates relations which can be constituted from three different classes (Nielson et al., 2004, pp.10–13);

1. The relation between the values of the function abstraction and their labels.
2. The relation between the values of variables and their labels.
3. The interrelations of application points and their relative function mappings:

   Application Point 1: The constraint representing the formal parameter of the function bounded with the actual parameter value.

   Application Point 2: The constraint representing the value outputted by said function.

There are multiple types of CFA analyses (Nielson et al., 2004, pp.139–195):

- Abstract 0-CFA Analysis,
- Syntax Directed 0-CFA Analysis,
- Constrain Based 0-CFA Analysis,
- Uniform *k*-CFA Analysis,

2.3.2 Dataflow Analysis

In Data Flow Analysis, the program is subdivided into sections via elementary blocks, connected by edges, describing the delegation of control in the program. There are two primary methodologies of approaching Data Flow analysis; The Equational Approach, and the Constraint Based approach (as mentioned in Section 2.3.1)

The equational approach extracts a number of equations from a program; belonging to the following classes:

(andre:) create figures for each one... number 1 is one to 1, number 2 is many to one... include source code

- The relation between the exit information of a node to the entry information of the same node (flow of data).
- The relation between the entry information of a node to exit information of nodes from which control could have possibly come from.

There are multiple types of Intra-procedural Data Flow analyses as may be seen below (Nielson et al., 2004, pp.33–51):

- Available Expression Analysis,
- Reaching Definition Analysis,
- Very Busy Expression Analysis,
- Live Variable Analysis.

An important form of analysis in this subsection is the Reaching Definition Analysis. It is made use of in other analyses, such as Abstract Interpretation. This type of analysis relates distinct labels to allow the identification of the primitive constructs of a program without the need to construct a flow graph.

2.3.3 Abstract Interpretation

This form of analysis is the way Analyses are calculated rather than how their specification is constructed. Thus, it is independent of the specification style. The analysis maps an initial state and a fixed-point from a concrete domain onto an abstract domain; enabling program properties to be decidable (2.8). Abstract Interpretation is a three-step procedure (Nielson et al., 2004, pp.13–17);

Collection of Semantics

This is the preliminary step which records a set of traces that can possibly reach a program point. A trace records the origins of a variables value. From a trace, a set of semantically reaching definitions can be extracted; pairs of variables and labels.

Galois Connections

A Galois connection is the joining of the *trace* sets and a *reaching definition* sets; creating a relation. The construction of this joint set is realized by an abstract function α and a concretisation function γ (seen in Figure 2.7), forming the set (α, λ). The abstraction function extracts reachability information present in a set of traces, whilst the concretisation function produces all traces which are consistent with the given reachability information (Nielson et al., 2004, pp.14–15).

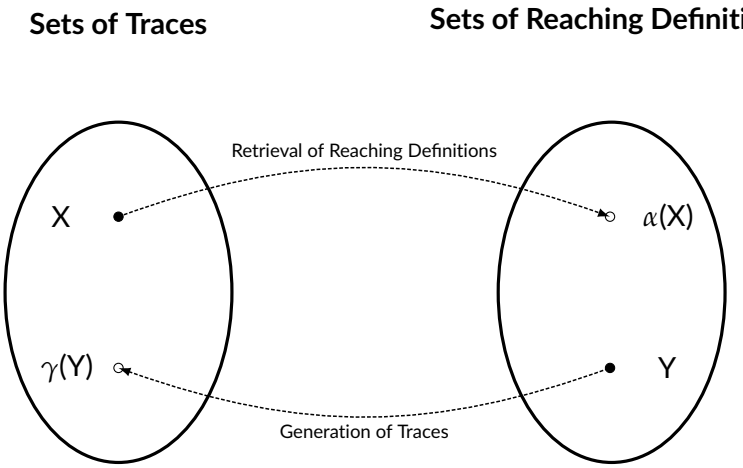

Figure 2.7: Realization of Galois Connection

Induced Analysis

Finally, an induced analysis is performed on the information obtained, providing a calculated analysis of the previously undecidable properties, as shown in Figure 2.8. This type of analysis provides a result which is produced efficiently, and relatively precisely.

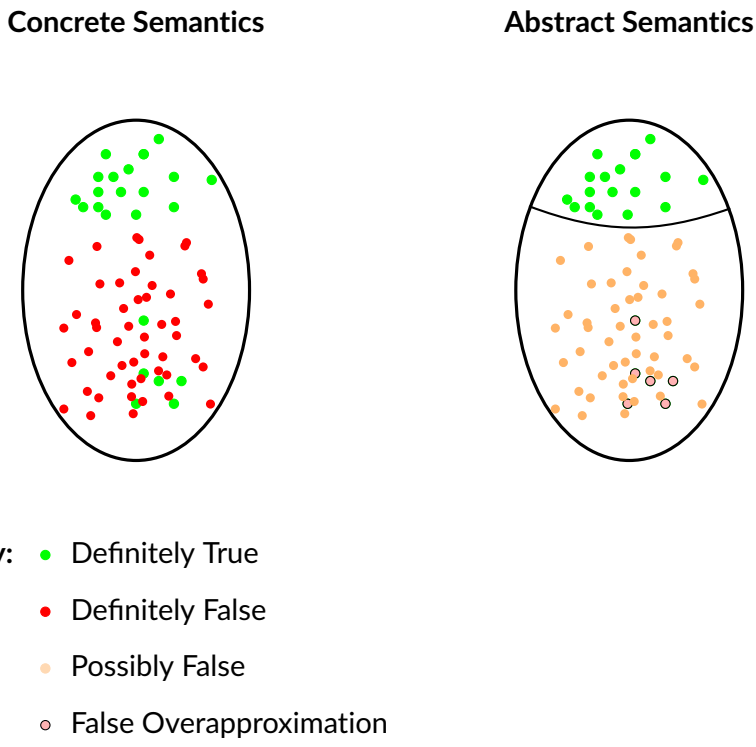

Figure 2.8: Efficiency-Precision trade-off presented by Abstraction Interpretation

2.3.4 Type and Effect Systems

Type and effect systems are the amalgamation of both an Effect System and an Annotated Type System (Nielson et al., 2004, pp.17–18). In an Effect System information about what happens when an execution occurs, rendering a change the current state, is produced (ex: what exception might be raised if this execution occurs). In an Annotated Type system the judgements that occur describe certain properties of states, such as a variables signum. Further detail into this method of analysis will not be delved into, as it is out of this papers scope. A simple Type and Effect listing may be seen in Listing 2.3.4.

(andre:) fix format+tabulation in listing

```
1     let val ref = reference (fn x=>x)   TYPE:   int -> int
  reference
2                         EFFECT: Creates an
3                             int -> int reference
4     in { ref := (fn n=>n+1);
5       !ref true
6       }
```

```
7       end
8
```

Listing 2.8: Type & Effect System example on SML Code Snippet

Chapter 3

Methodology

This chapter describes how the Proposed Solution was implemented, and why certain decisions were taken regarding design implementation. This chapter also explores the problems encountered whilst formulating such a design.

3.1 Initial Challenges

3.2 Abstract Design

PATH is a three-phase bytecode inspection tool (Sections 3.2.1 & 3.3 respectively). The high-level implementation is as follows; PATH iteratively inspects each bytecode instruction produced by the Python Compiler, generating facts as instructions are iterated through.

3.2.1 Phase Overview

1. The preliminary phase is the setup phase. This phase tackles the initialization of all the global variables required for bytecode inspection along with the statement relations.

2. The second phase is the iteration phase; AIL. This phase is where the main logic of the framework is found. The manipulation of local and global variables, pertaining to the IR and fact generation also occurs in this phase.

3. The third and final phase generates a block summary, by retrieving information from the generated facts in phase 2.

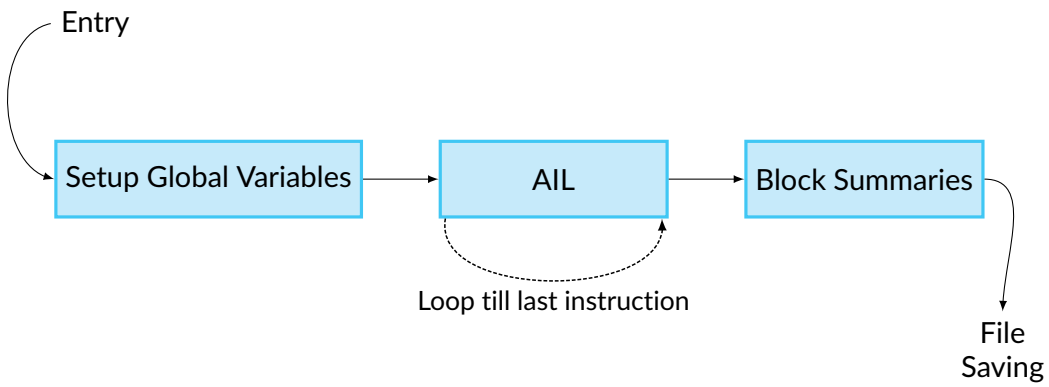

Figure 3.1: Phases of PATH

3.3 Concrete implementation

3.3.1 Setup

The setup stage is the entry point to the framework. This stage is simple; setting up of the basic variables, which provide a basis for the upcoming logic of PATH.

Relation Initialization

The first stage deals with the initialization of the *Statement Relations*; imperative for fact generation. Most of the relations are initially realized as sets. Lists are only used for the relations which are required to be stored in an ordered fashion. This requirement is motivated by the relations dependency, so as to achieve proper internal fact generation. These relations are `simple_statement_ir`, `statement_uses_local`, and `statement_uses_global`. By design, as one reaches the termination of PATH, these relations are consolidated into a Python dictionary; returned as `fact_dict`. An in-depth review of all relations is found in Section 3.4

Global Variable Initialization

Following the initialization of the relations, the *Global Variables* are conceived. These variables are used throughout PATH, namely for proper block and instruction variable handling, thus are divided into two main distinct groups; Global Instruction Variables, and Global Block Variables. Any other variables are classified as Miscellaneous Variables. All Global Variables are initialized as empty variables, unless said otherwise.

3.3.2 AIL

An abundant amount of the framework's logic is packed into the AIL. This loop iterates through every bytecode instruction, recording attributes of the instruction, and how each instruction interacts with the previous instruction, and will interact with the next instruction. The iterative nature (known as III) of the AIL was inspired by the Python VM Innards workings (See Figure 2.3). Inspecting the bytecodes in such a fashion (i.e. generating all the facts iteratively) is arguably faster than having a phase per fact needed to be generated. In addition to fact generation, we also incorporated CFA and IR generation in the III fashion; taking a different approach to how current inspection tools operate (such as Pylint), whereby first the IR is generated, following an analysis conducted on the IR itself. This is a multiphase process, in comparison to the single phase design created by AIL. The AIL is visualised in Figure 3.2.

Bytecode Validity

Firstly, bytecode instruction validity is ensured via the specified opcode dictionary (supported Bytecode Instructions can be found in Opcode Table).

Local Variable Initialization

Following verification, local instruction variables are set. These variables are the first to be set as they are the basis for fact generation and block generation. These variables are also the foundation for the IR generated, namely for setting the instruction line number and the instruction identifier. The former is dynamically set, dependent on the amount of opcodes that need to be processed, whilst the latter is generated by a specially designed MD5 algorithm, creating parameter dependent unique hashes for every instruction.

Block Handling

Block handling is the subsequent step, modifying the local block variables. In PATH the notion of a [elementary] block is a primitive from which a program is constructed by. Entry and exit points of blocks depend on the current program flow (as seen in Section 2.3.2); they form part of the basis of both DFA and CFA. New blocks are uniquely identified (similarly to the instruction identifier) and instructions are bound to their respective block (as is seen in Section 3.4). By design, each block has its own stack and unique properties (see Section 3.5) that are used as a basis for both fact generation and accurate block analysis.

Figure 3.2: AIL Innards

An interesting design feature of AIL is the incorporation of both Block Handling and Control Flow Analysis in one sub-phase. Moreover, this sub-phase is only executed if an instruction forms part of a new block; avoiding redundant execution stages.

Opcode Handling

The penultimate stage handles general opcode tasks (i.e: general relation generation) and opcode specific tasks (i.e: recursive `MAKE_FUNCTION` dictionary nesting). There exist opcodes, such as `MAKE_FUNCTION` which require additional parameters to be considered for accurate fact generation. The incorporation of the opcode specific IR generation in this stage, expedites the process of fact and IR generation, compared to having the IR generated in a separate stage.

Updating Global Variables

The final stage in the AIL simply updates the global variables that are to be used in the next iteration.

3.3.3 Block Summary

The termination of PATH is brought about by the block summaries, conducted on the facts generated in the AIL. A block summary is generated for every block, showing where blocks start and end; useful for external debugging purposes and instruction grouping. The generation of a block summary is a form of block analysis.

3.4 Fact Generation

In PATH a fact is information which is generated from the program that is being inspected. These facts are useful for end users as they provide deeper insight in the operations that occur in the program, reducing the overall engineering complexity of program analysis. PATH produces three types of facts: Statement facts, Block facts, and Program facts. Facts are generated in III fashion (per bytecode instruction) and outputted to the user as *.fact* files. A table of all the facts and their descriptions is found in the Facts Table.

3.5 Control Flow

Control Flow in PATH is implemented within the block handling stage in III fashion, as mentioned in Section 3.3.2. The block handling stage is broken down in three main parts; block variable generation, control flow handling and block I/O.

Block Variable Generation

Block variables are only generated if the start of a new block is detected. New blocks are based on jumps and labels, whereby a new block starts after a jump instruction or at a label. The latter is a jump target (i.e: which bytecode a jump instruction would jump to) and the former is self-explanatory. The detection of a new block triggers the block variables to be generated along with certain facts pertaining to block information.

```
1            if x>3:
2                #do something#
3            else:
4                #do something else#
5
```

Listing 3.1: Conditional statement script

```
1            0 LOAD_GLOBAL              0 (x)
             2 LOAD_CONST               1 (3)
             4 COMPARE_OP               4 (>)
             6 POP_JUMP_IF_FALSE       10

2            8 JUMP_FORWARD             0 (to 10)

4      >>   10 LOAD_CONST               0 (None)
12           RETURN_VALUE
```

Listing 3.2: Bytecode Dissasmbly of Listing 3.5

Control Flow Generation

Following the initialization, links between the edges of basic blocks are generated; as shown in Figure 3.3. It is ensured that upon encountering a jump, no redundant relations are created between blocks; taking care to only create relations between linked block

edges (i.e: a block's *if* block cannot enter the same *else* block). These type of relations are avoided by creating a unique identifier (`block_link_id`); uniquely identifying a link between two blocks.

3.6 IR Generation

The IR that PATH creates is intended to reduce the engineering complexity for further program analysis. The complex task of recording variables that are consumed and created in a more concise form was needed, so as to resolve the tedious task of manually keeping note of all stack operations. PATH offers this functionality for two primary operation subtypes:

1. Binary Operation Binding
2. Function Calling Binding

Providing a standardized representation for the cases above greatly simplifies the final representation of a program. IR generation takes a III approach; following a sequential iterative generational pattern.

Binary Operation Binding takes the arguments from the operation, binding them to a variable, summarizing a binary operation as follows:

```
<inst_id> <var_id> = <inst_name> <arg1> <operation> <arg2>
```

Listing 3.3: Binary Operation Binding Syntax

Function Calling Binding takes the function and arguments passed to the said function, binding them to the function identifier, indicated by a succeeding `STORE_FAST` instruction. This is shown below.

```
<inst_id> <func_id> = CALL_FUNCTION <arg1> ... <argN>
```

Listing 3.4: Function Calling Binding

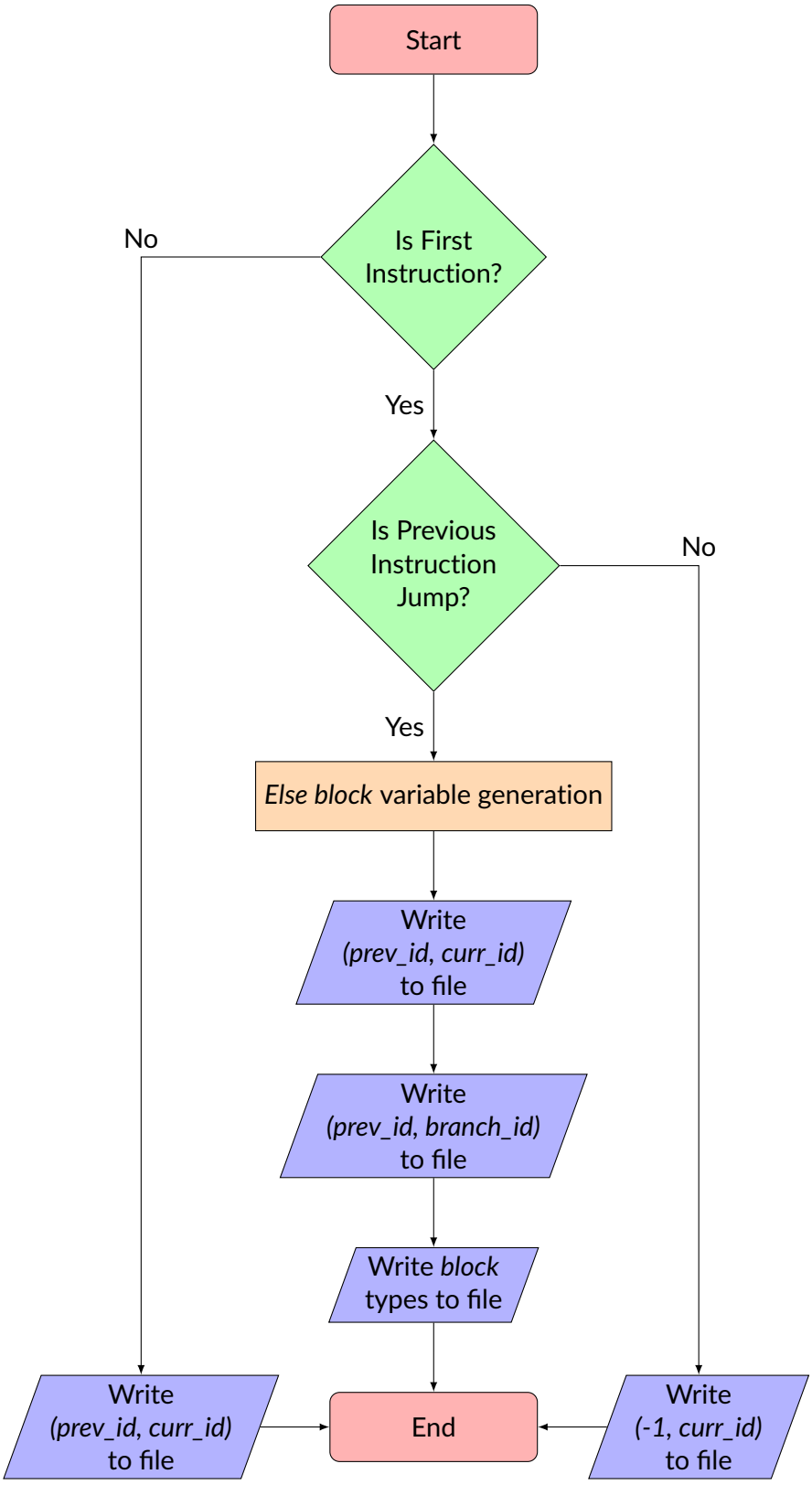

Figure 3.3: Control Flow Handling

3.7 Technical Implementation

Global Instruction Variables monitor several attributes of the individual bytecode instructions:

prev_instruction keeps track of instruction $_{i-1}$. Previous instructions are useful as they allow for proper block creation, and are vital for the generation of control flow graphs.

prev_instruction_identifier records the previous instruction identifier. The identifier is stored in a separate variable, and not extracted from `prev_instruction` for the sake of brevity. Variable is initialized to **-1**, indicating the start of the program.

instruction_list produces a list of all the bytecode instructions that will be enumerated through in the AIL. This list is immediately created to ease the creation of other metrics, such as obtaining the total amount of instructions.

next_instruction pulls the next instruction from `instruction_list`, if there is a next instruction. This variable is used for correct block handling and IR generation.

instruction_size finds the amount of bytecode instructions there are in the current scope, that need to be read. Variable is initialized.

instructions_offset_list stores all the offsets of the bytecode instructions. It is interesting to note, that through this dissertation it has been discovered that bytecode instructions are always 2 bytes in size. Each byte is reserved for the opcode and oparg respectively.

largest_bytecode_offset stores the largest bytecode offset. This represents the last instruction, and is used to generate an upper limit for the custom line number argument. Variable is initialized.

line_number monitors the current line number argument of each bytecode instruction. Bytecode instructions do not always have a line number, indicating that the instruction is between line numbers. When an instruction is in such a state, it relies on the custom line number argument.

Global Block Variables perform a vital role in distinguishing blocks; useful for block analysis.

block_identifier keeps track of the current instruction block.

prev_block_identifier records the previous block identifier. Variable is initialized to **-1**, indicating the first block of the program.

The following variables form part of the Miscellaneous Variables. These form part of the backbone structure of PATH:

frame_stack acts as a frame stack, having instructions pushed onto and popped off of accordingly. This stack is flushed with every new function call.

prev_frame_stack records the previous frame stack, from a previous function call.

block_stack acts as a stack, having instructions pushed onto and popped off of accordingly. This stack is flushed with every new block. An interesting insight that was discovered is that Python operates in a way whereby prior to exiting a block, no instructions are left on the stack.

prev_block_stack records the previous block stack.

tos stores the value at the top of the stack. Variable is used for convenience.

bytecode stores the disassembled code object.

Chapter 4

Evaluation

In this chapter PATH is tested and evaluated. For testing, programs with varying complexity are used; starting from simple programs ranging to more complex programs. There are three main Research Questions, which will be put to the test in Sections 4.3, 4.4 and 4.5; validating PATH.

4.1 Experimental Results

Programs from an increasing range of complexity were tested in this section. Individual opcode features of PATH were verified with the simple functions found in *project/tests/opcode*, logical functionality of PATH (such as CFG generation) was tested with functions found in *project/tests/logic*, and finally, general purpose testing was conducted via open source programs[1]. All functions passed the Fact Generation, Control Flow and IR Generation stages, reporting satisfactory results. Performance results covering all functions can be found in Figure 4.1.

4.2 Research Questions

The research questions below aid in systematically investigating the need and use of PATH. These questions are delved into more depth in Sections 4.3, 4.4, and 4.5.

[1] obtained from `https://github.com/OmkarPathak/Python-Programs/tree/master/Programs`

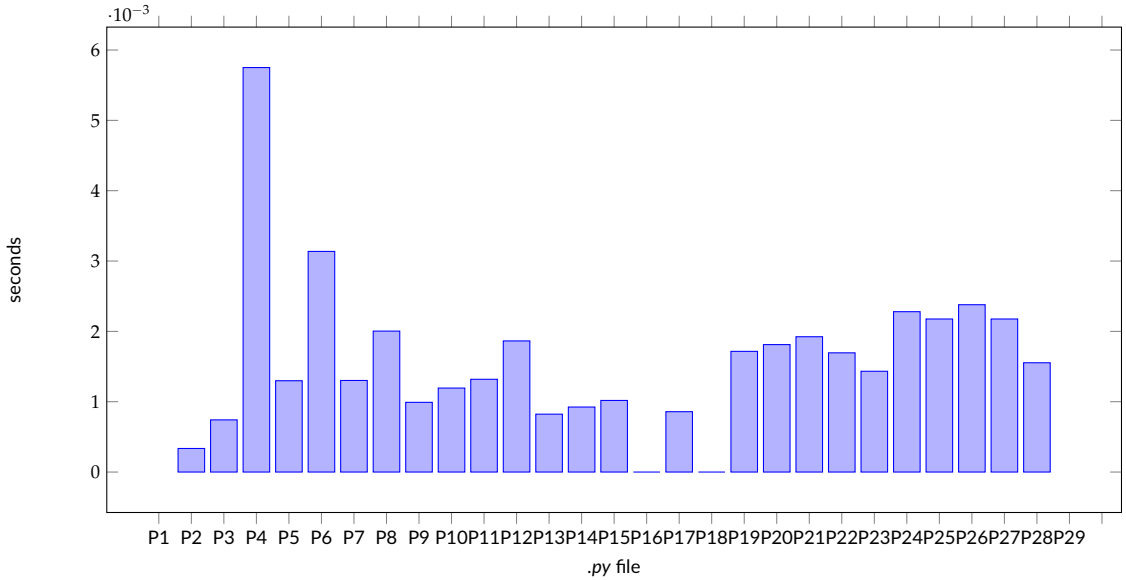

Figure 4.1: Performance Metrics of PATH

4.2.1 How further analyses is facilitated

PATH creates *.facts* files. Such files contain metrics pertaining to the function analysed that are of interest to end users. These files' contents are tabulated in Facts Table.

4.2.2 CPython invariants, and findings

Throughout this project, several undocumented findings have been concluded from results generated by PATH processing the functions in *project/tests*. The primary undocumented findings are listed below:

- Elementary Blocks in CPython do not retain any content on their frame stacks.
- `LOAD_GLOBAL` is reserved for storing function names.
- Bytecode operations on the frame stack.

These findings amongst others are further discussed in Section 4.4.

4.2.3 Scalability of PATH

For PATH to be of any practical use, it must be able to scale appropriately, and traverse through different function calls. This is possible as it follows a recursive methodology when dealing with `MAKE_FUNCTION` calls. PATH

4.3 Facilitating further analyses

Program analysis is a computationally intensive and time-consuming process. This process is facilitated by the use of automated tools, such as PATH. The *facts* generated by PATH are thought out in such a way so as to analyse functions and to enable any further analyses one might conduct. A prime example would be the potential to carry out a Data Flow Analysis with the information generated by PATH. For a Data Flow Analysis one requires a CFG (generated as mentioned in Section 3.5), and data-flow equations for every node of the CFG. A popular approach to Data Flow Analysis is the Reaching Definitions Method (See section 2.3.2). This would be facilitated with PATH via the CFG generated and the following *.facts* files: *StatementUsesLocal.facts* and *PushValue.facts* (refer to Facts Table); demonstrating that it does indeed reduce the engineering complexity of further possible analysis.

This tool enables teams to focus on the analysis itself, negating the need of the preliminary step for the creation of different intermediary representations to conduct analysis on. The fact that there are not many professionals in this area makes complex analysis tools expensive and rare to come across.

4.3.1 Relating PATH with current frameworks

In reality, program analysis cannot be simply bisected into two categories; it is more accurately depicted by the spectrum shown in

Similar Frameworks

PATH is a pure basic program analysis framework, in which its stereobate lies with the generation of an IR. This framework takes inspiration from several other Static frameworks which operate similarly:

doop A Java pointer and Taint Analysis framework that conducts analysis by the Soufflé Datalog Engine (Bravenboer and Smaragdakis, 2009).

soot A Java optimization framework, providing analyses ranging from CFG construction to Taint analysis (Vallée-Rai et al., 2010).

Gigahorse An Ethereum analysis framework, specializing in the decompilation of smart contracts (Grech et al., 2019).

Vandal A fast and robust security analysis framework for Ethereum smart contracts (Brent et al., 2018).

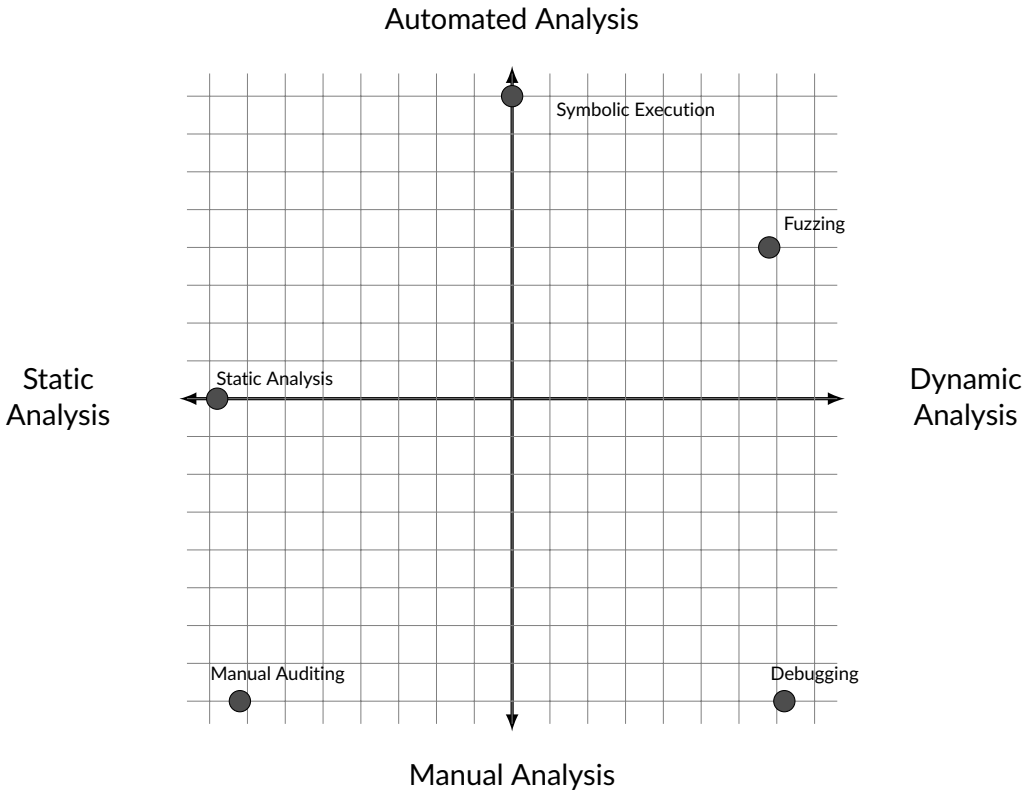

Figure 4.2: Spectrum of Program Analysis

The frameworks mentioned above are **pure** program analysis frameworks, whereby the analysis is conducted on an IR.

Different Frameworks

4.4 Insights

4.5 Scalability Experiments

4.6 Case Study

Chapter 5

Conclusion

Chapter 6

Conclusions

This section should have a summary of the whole project. The original aims and objective and whether these have been met should be discussed. It should include a section with a critique and a list of limitations of your proposed solutions. Future work should be described, and this should not be marginal or silly (e.g. add machine learning models). It is always good to end on a positive note (i.e. 'Final Remarks').

6.1 Revisiting the Aims and Objectives

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

6.2 Critique and Limitations

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain

all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

6.3 Future Work

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

6.4 Final Remarks

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

References

Andersen, L. O. *Program analysis and specialization for the C programming language*. PhD thesis, Citeseer, 1994.

Aycock, J. Converting python virtual machine code to c. In *Proceedings of the 7th International Python Conference*, pages 76–78, 1998.

Bacon, J. W. The stack frame, 2011. URL `http://www.cs.uwm.edu/classes/cs315/Bacon/Lecture/HTML/ch10s07.html`.

Bennett, J. An introduction to python bytecode, 2018. URL `https://opensource.com/article/18/4/introduction-python-bytecode`.

Bolz, C. F., Cuni, A., Fijalkowski, M., and Rigo, A. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25, 2009.

Bonta, V. and Janardhan, N. K. N. A comprehensive study on lexicon based approaches for sentiment analysis. *Asian Journal of Computer Science and Technology*, 8(S2):1–6, 2019.

Bravenboer, M. and Smaragdakis, Y. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 243–262, 2009.

Brent, L., Jurisevic, A., Kong, M., Liu, E., Gauthier, F., Gramoli, V., Holz, R., and Scholz, B. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981*, 2018.

Conrad, P. Python function calls and the stack lesson 2, 2010. URL `https://sites.cs.ucsb.edu/~pconrad/cs8/topics.beta/theStack/02/`.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms*. The MIT Press, 3rd edition edition, 2009.

Developers, B. Bandit, 2022. URL `https://bandit.readthedocs.io/en/latest/`.

Forcier, J., Bissex, P., and Chun, W. J. *Python web development with Django*. Addison-Wesley Professional, 2008.

Foundation, P. S. dis-disassembler for python bytecode, 2022a. URL `https://docs.python.org/3/library/dis.html#dis.Bytecode`.

Foundation, P. S. Code objects, 2022b. URL `https://docs.python.org/3/c-api/code.html`.

Geurts, L., Meertens, L., and Pemberton, S. *ABC programmer's handbook*. Prentice-Hall, Inc., 1990.

Goldsborough, P. A tour of tensorflow. *arXiv preprint arXiv:1610.01178*, 2016.

Grech, N., Brent, L., Scholz, B., and Smaragdakis, Y. Gigahorse: thorough, declarative decompilation of smart contracts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1176–1186. IEEE, 2019.

Index, T. The python programming language, 2022. URL https://www.tiobe.com/tiobe-index/python/.

Intel. Dynamic analysis vs. static analysis, 2013. URL https://www.cism.ucl.ac.be/Services/Formations/ICS/ics_2013.0.028/inspector_xe/documentation/en/help/GUID-E901AB30-1590-4706-94B1-9CD4736D8D2D.htm.

Juneau, J., Baker, J., Wierzbicki, F., Muoz, L. S., Ng, V., Ng, A., and Baker, D. L. *The definitive guide to Jython: Python for the Java platform*. Apress, 2010.

Landin, P. J. The mechanical evaluation of expressions. *The computer journal*, 6(4):308–320, 1964.

Logilab, P. Pylint, 2022. URL https://pylint.pycqa.org/en/latest/.

Lutz, M. *Programming python*. " O'Reilly Media, Inc.", 2001.

Møller, A. and Schwartzbach, M. I. Static program analysis. *Notes. Feb*, 2012.

Mueller, J. P. *Professional IronPython*. John Wiley & Sons, 2010.

Nielson, F., Nielson, H. R., and Hankin, C. *Principles of program analysis*. Springer Science & Business Media, 2004.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

Rice, H. G. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society*, 74(2):358–366, 1953.

Rossum, G. V. et al. python/cpython, 2022. URL https://github.com/python/cpython/blob/d93605de7232da5e6a182fd1d5c220639e900159/Python/ceval.c#L716.

Saabith, A. S., Fareez, M., and Vinothraj, T. Python current trend applications-an overview. *International Journal of Advance Engineering and Research Development*, 6(10), 2019.

Shaw, A. Your guide to the cpython source code, 2022. URL https://realpython.com/cpython-source-code-guide/.

Srinath, K. Python–the fastest growing programming language. *International Research Journal of Engineering and Technology (IRJET)*, 4(12):354–357, 2017.

Summerfield, M. *Rapid GUI Programming with Python and Qt: The Definitive Guide to PyQt Programming (paperback)*. Pearson Education, 2007.

Sweigart, A. *Making Games with Python & Pygame*. 2012.

Thomas, D. Benefits of dynamic typing. Online, 2013. URL https://wiki.c2.com/?BenefitsOfDynamicTyping.

Tismer, C. Continuations and stackless python. In *Proceedings of the 8th international python conference*, volume 1, 2000.

Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. 2010.

van Rossum, G. Python (programming language) 1 cpython 13 python software foundation 15.

Van Rossum, G. Python reference manual. *Department of Computer Science [CS]*, (R 9525), 1995.

van Rossum, G. and de Boer, J. Interactively testing remote servers using the python programming language. *CWi Quarterly*, 4(4):283–303, 1991.

Van Rossum, G. and Drake Jr, F. L. Python reference manual, 1994.

Van Rossum, G. and Drake Jr, F. L. *Python tutorial*, volume 620. Centrum voor Wiskunde en Informatica Amsterdam, The

Netherlands, 2020.

Van Rossum, G. et al. Python programming language. In *USENIX annual technical conference*, volume 41, pages 1–36, 2007. URL `https://www.wikizero.com/en/Python_(language)`.

Zorn, B. *Barrier methods for garbage collection*. Citeseer, 1990.

Appendix A

Opcode Table

Opcode List			
Class	Opname	Push Value	Pop Value
GENERAL OPERATIONS	NOP	0	0
	POP_TOP	0	1
	ROT_TWO	0	0
	ROT_THREE	0	0
	ROT_FOUR	0	0
	DUP_TOP	1	0
	DUP_TOP_TWO	2	0
UNARY OPERATIONS	UNARY_POSITIVE	0	0
	UNARY_NEGATIVE	0	0
	UNARY_NOT	0	0
	UNARY_INVERT	0	0
	GET_ITER	0	0
	GET_YILED _FROM_ITER	0	0
BINARY OPERATIONS	BINARY_POWER	1	2
	BINARY_MULTIPLY	1	2
	BINARY_MATRIX _MULTIPLY	1	2
	BINARY_FLOOR _DIVIDE	1	2
	BINARY_TRUE _DIVIDE	1	2

	BINARY_MODULO	1	2
	BINARY_ADD	1	2
	BINARY_SUBTRACT	1	2
	BINARY_SUBSCR	1	2
	BINARY_LSHIFT	1	2
	BINARY_RSHIFT	1	2
	BINARY_AND	1	2
	BINARY_XOR	1	2
	BINARY_OR	1	2
INPLACE OPERATIONS	INPLACE_POWER	0	1
	INPLACE_MULTIPLY	0	1
	INPLACE_MATRIX _MULTIPLY	0	1
	INPLACE_FLOOR _DIVIDE	0	1
	INPLACE_TRUE _DIVIDE	0	1
	INPLACE_MODULO	0	1
	INPLACE_ADD	0	1
	INPLACE_ADD	0	1
	INPLACE_SUBTRACT	0	1
	INPLACE_LSHIFT	0	1
	INPLACE_RSHIFT	0	1
	INPLACE_AND	0	1
	INPLACE_XOR	0	1
	INPLACE_OR	0	1
COROUTINE OPERATIONS	GET_AWAITABLE	0	3
	GET_AITER	0	0
	GET_ANEXT	1	0
	END_ASYNC_FOR	(andre:) check	(andre:) check
	BEFORE_ASYNC_WITH	1	0
	SETUP_ASYNC_WITH	(andre:) CHECK	(andre:) CHECK
MISC. OPERATIONS	GET_AWAITABLE	0	1
	PRINT_EXPR	0	1

	LIST_APPEND	0	1
	MAP_ADD	0	2
	RETURN_VALUE	0	1
	YIELD_VALUE	0	1
	YIELD_FROM	0	1
	YIELD_FROM	0	0
	SETUP_ANNOTATIONS	0	0
	IMPORT_STAR	0	1
	POP_EXCEPT	0	3
	RERAISE	0	0
	WITH_EXCEPT_START	1	0
	LOAD ASSERTION ERROR	1	0
	LOAD BUILD CLASS	1	0
	GET_LEN	1	0
	MATCH_MAPPING	1	0
	MATCH_SEQUENCE	1	0
	MATCH_KEYS	1	0
	STORE_SUBSCR	0	3
	DELETE_SUBSCR	0	2
ARGUMENT OPERATIONS	STORE_NAME	0	1
	DELTE_NAME	0	0
	UNPACK_SEQUENCE	**ARG**	1
	UNPACK_EX	0	1
	STORE_ATTR	0	2
	DELETE_ATTR	0	1
	STORE_GLOBAL	0	1
	DELETE_GLOBAL	0	0
	LOAD_CONST	1	0
	LOAD_NAME	1	0
	BUILD_TUPLE	1	**ARG**
	BUILD_LIST	1	**ARG**
	BUILD_SET	1	**ARG**
	BUILD_MAP	1	**ARG**

BUILD_CONST _KEY_MAP	1	**ARG**
BUILD_STRING	1	**ARG**
LIST_TO_TUPLE	1	1
LIST_EXTEND	0	1
SET_UPDATE	0	1
DICT_MERGE	0	1
LOAD_ATTR	0	0
COMPARE_OP	0	1
IS_OP	0	1
CONTAINS_OP	0	2
IMPORT_NAME	0	1
IMPORT_FROM	0	1
JUMP_FORWARD	0	1
POP_JUMP_ IF_TRUE	0	1
POP_JUMP_ IF_FALSE	0	1
JUMP_IF_ NOT_EXC_MATCH	0	1
JUMP_IF_ TRUE_OR_POP	0	**ARG**
JUMP_IF_ FALSE_OR_POP	0	**ARG**
JUMP_ABSOLUTE	0	0
FOR_ITER	0	(andre:) CHECK
LOAD_GLOBAL	1	0
LOAD_FAST	1	0
STORE_FAST	0	1
DELETE_FAST	0	0
LOAD_CLOSURE	1	0
LOAD_DEREF	1	0
LOAD_CLASSDEREF	1	0
STORE_DEREF	0	1
DELETE_DEREF	1	0

	RAISE_VARARGS	0	**ARG**
	CALL_FUNCTION	1	**ARG**
	CALL_FUNCTION_KW	0	1
	CALL_FUNCTION_EX	0	1
	LOAD_METHOD	2	1
	CALL_METHOD	1	**ARG**
	MAKE_FUNCTION	1	(andre:) check
	BUILD_SLICE	0	**ARG**
	EXTENDED_ARG	0	0
	FORMAT_VALUE	1	(andre:) check
	MATCH_CLASS	0	2
	GEN_START	0	1
	ROT_N	0	0

Appendix B

Facts Table

Facts List	
.facts file	Description
BlockInputContents.facts	(Block_ID: `int`, Input_Active_Statment_ID: `int`)
BlockOutputContents.facts	(Block_ID: `int`, Output_Active_Statement_ID: `int`)
BlockSummary.facts	Block_ID: `int`, Start_Statement_ID: `int`, End_Statement_ID: `int`, Start_Instruction_Offset: `int`, End_Instruction_Offset: `int`
BlockToBlock.facts	(Block_ID: `int`, Next_Block_ID: `int`)
BlockType.facts	(Block_ID: `int`, Block_Type: `int`, Block_Link_ID: `int`)
PushValue.facts	SET[(Statement_ID: `int`, Value: `value`)]
SimpleIR.facts	See IR Generation
StatementBlock.facts	SET[(Statement_ID: `int`, Block_ID: `int`)]
StatementBlockHead.facts	(Statement_ID: `int`, Block_ID: `int`)
StatementBlockStackSize.facts	(Statement_ID: `int`, Stack_Size: `int`)
StatementBlockTail.facts	(Statement_ID: `int`, Block_ID: `int`)
StatementCode.facts	SET[(Statement_ID: `int`, CodeObject: `code`)]

StatementDefinesLocal.facts	
StatementDetails.facts	SET[(Statement_ID: `int`, Block_ID: `int`, Push_Value: `value`), Opname: `String`, Line_ID: `float`)]
StatementMetadata.facts	SET[(Statement_ID: `int`, Line_ID: `float`)]
StatementNext.facts	SET[(Statement_ID: `int`, Next_ID: `int`)]
StatementOpcode.facts	SET[(Statement_ID: `int`, Opname: `String`)]
StatementPopDelta.facts	SET[(Statement_ID: `int`, Pop_Delta: `int`)]
StatementPops.facts	SET[(Statement_ID: `int`, Pop_No: `int`)]
StatementPushes.facts	SET[(Statement_ID: `int`, Push_No: `int`)]
StatementUsesLocal.facts	SET[(Statement_ID: `int`, Used_Statement_ID: `int`)]
TotalStatementPopDelta.facts	(Statement_ID: `int`, Running_Pop_Count: `int`)