Program Analysis: Towards the Analysis of CPython Bytecode

André Theuma

Supervisor: Dr. Neville Grech

Faculty of ICT

University of Malta

22/04/2022

Submitted in partial fulfillment of the requirements for the degree of B.Sc. I.C.T. (Hons.)

Faculty of ICT

Declaration

I, the undersigned, declare that the dissertation entitled:

Program Analysis: Towards the Analysis of CPython Bytecode

submitted is my work, except where acknowledged and referenced.

André Theuma

27/05/2022

Acknowledgements

your acknowledgements

Abstract

(andre:) Reworked 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.

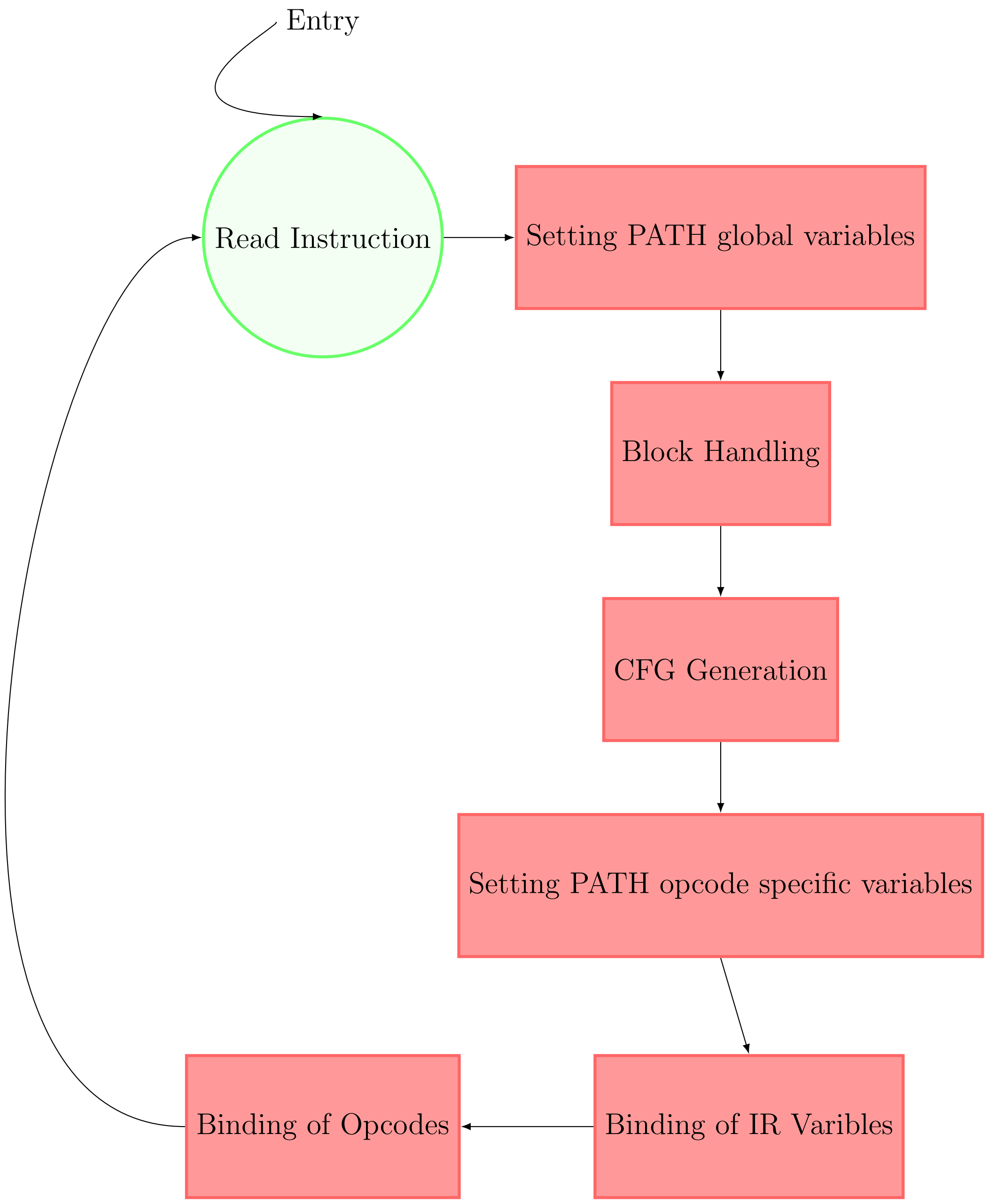

Figure 1: PATH High-Level Overview

Contents

List of Figures

List of Tables

1. Introduction

(andre:) Reworked Introduction

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 for software systems; allowing further development and refinement of the said system. Core vulnerabilities may be exposed and optimizations can be implemented. Tools such as Pylint [?], and Bandit [?] already exist in the industry.

PATH (Python Analysis Tooling Helper) 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.

1.1 Preliminary Overview

1.1.1 Python & CPython

(andre:) Reworked

Python is a high-level, object-oriented scripting language [?], suited for a wide range of development domains; from text processing [?] to machine learning [?] to game development [?]. The language's wide adoption (TIOBE's Language of the Year: 2007, 2010, 2018, 2020, 2021; [?]) may be attributed to the fact that it is based on the English language [?], making it easy to learn; aiding in the production of relatively complex programs. It is used extensively for rapid prototyping, as well developing fully-fledged real-world applications.

The most common and comprehensive implementation of Python is known as CPython [?]; a bytecode interpreter for Python, written in C.

1.1.2 Program Analysis

(andre:) Reworked

Complex programs imply complex behaviours. Such behaviours have to be analysed, as they might highlight certain vulnerabilities, and possibly indicate where optimizations can be carried out in the program. This area of interest is known as Program Analysis. Program Analysis provides answers for 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 properly conducting program analysis on these systems is ever-growing. There are two main approaches regarding 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 [?]. Since all (significant) properties of the behaviour of programs written in today's' programming languages are mathematically undecidable [?], 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 [?]; being the main motivation behind Static analysis. With the right kind of approximations, Static analysis provides guarantees regarding 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.

1.2 Aims & Objectives

(andre:) Reworked

The essential aim of this dissertation is the production of an analytical tool (PATH) which can be easily implemented in existing software systems. The tool is to be used on Python V3.10 [?] systems which are interpreted with the CPython [?] 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 being analysed. The standardized IR that is generated has to summarize the block analysis that is done by PATH.

The creation of this tool is vital as there is a lack of security and analysis tooling, for what is the worlds most used programming language [?]; giving developers an increased freedom of choice when having to choose an analysis framework for their projects.

1.3 Chapters Overview

This paper is composed of five chapters, which can be broken down in the following manner:

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

Chapter 2 gives an in-depth literature review of the content that is 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 the methodology and implementation of the disassembler

(PATH). The design of choice is further discussed together with the reasons for ongoing into such a design are highlighted.

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 further work that can be done.

(andre:) New

1.4 Contributions

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 ??.

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.

2. Background & Literature Review

2.1 Python

2.1.1 History

Python being a general-purpose [?], high-level programming language [?, 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 [?].

The language was developed to be a successor of the ABC programming language [?] and was 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 that is intended to be used in a more professional environment, whilst ABC is geared towards non-expert computer users [?, 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 [?]. It was designed with the intention 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 [?, pp.4–5]) which is unusual among other popular programming languages [?, pp.2–3]; new blocks increase in indentation, whilst the end of a current block is signified by the decrease of an indentation. 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 [?]. 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 [?]; appealing to a wider audience. Python also includes a cycle-detecting garbage collector [?], freeing up objects as soon as they become unreachable [?, pp.9–10]. Objects are not explicitly freed up as the collector requires a significant processing overhead [?, 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++ [?] 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 [?] to complex Machine Learning frameworks [?], further increasing ease-of use, and supporting the quick and simple development philosophy of Python.

2.1.3 Implementations

There are several environments which support the Python language; known as implementations. The default, most feature comprehensive Python implementation is CPython[2] [?] ,written and maintained by Guido van Rossum. Other popular re-implementations include PyPy [?], Jython [?] and IronPython [?]. This paper will focus on the CPython implementation of the Python language and will not cover

[1]Over 329,000 packages as of September 2021 [?]

[2]The terms CPython and Python are typically used interchangeably. This will be the case in this paper, unless specified otherwise.

any of the other alternate implementations.

2.2 CPython

2.2.1 Overview

(andre:) Reworked

CPython is the predominant implementation of the Python language, and is written in C. It has a thin compilation layer from source code to CPython bytecode (See **??**); simplifying the design of the interpreter. Unlike the typical structured program representations, bytecode is easier to parse, and has a standardized notation.

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

[3] *ceval.c*

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

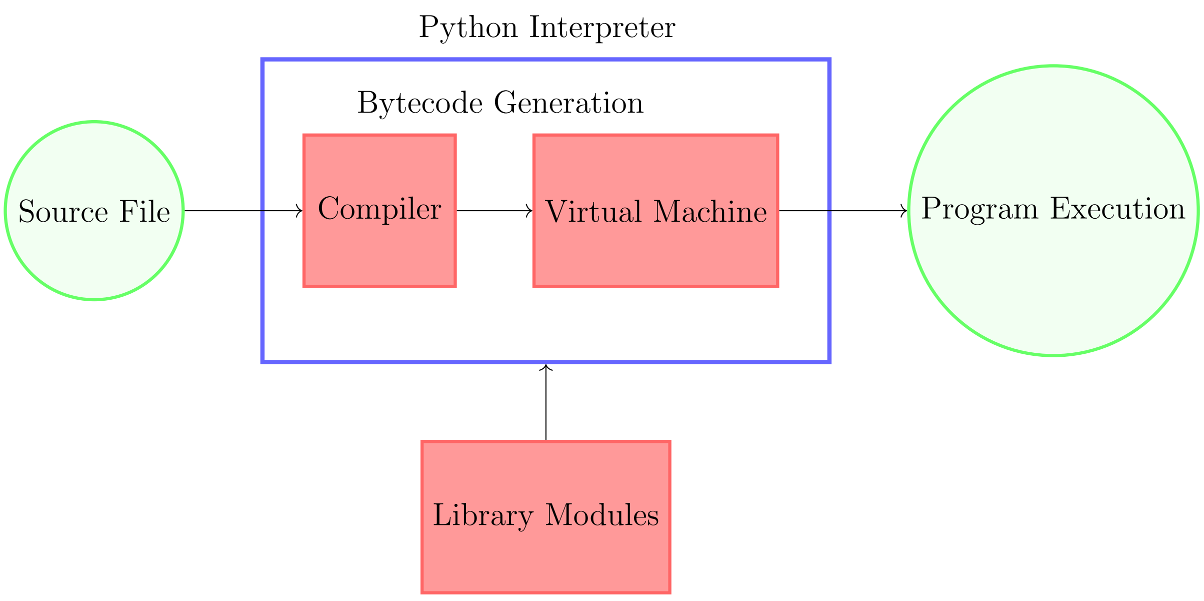

Figure 2.1: Python Code Execution

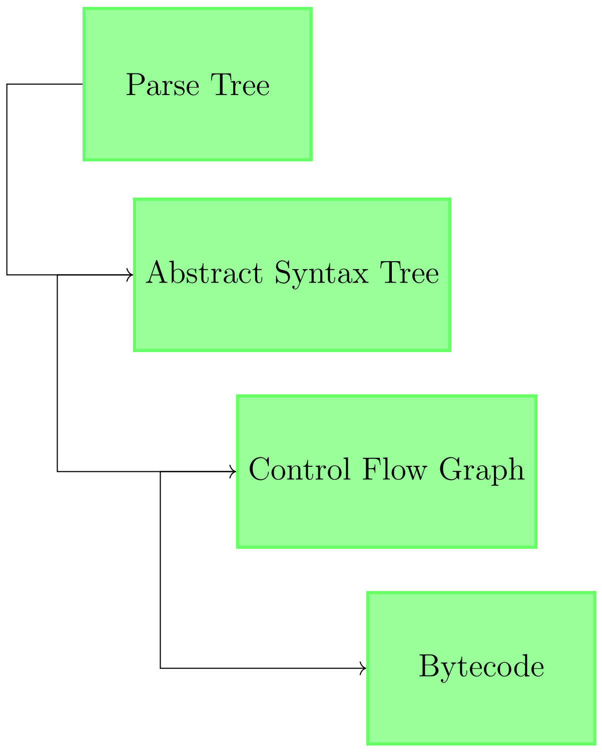

Figure 2.2: Compiler Innards

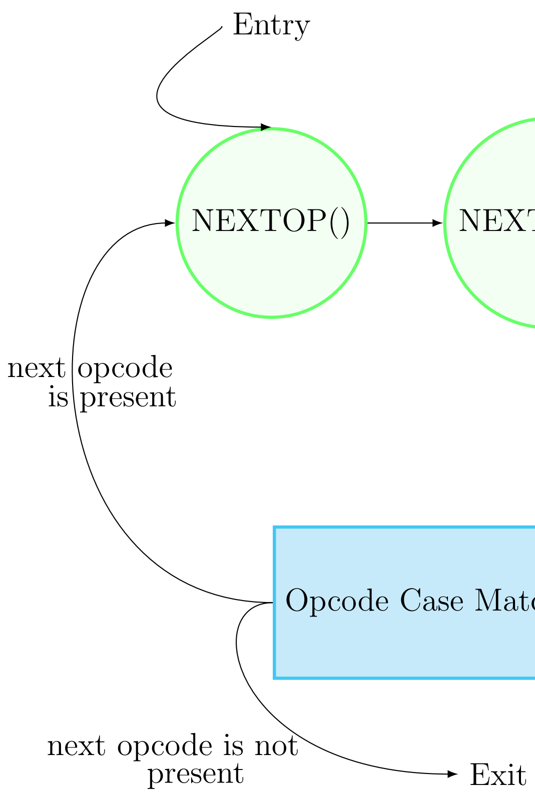

Figure 2.3: VM Innards

This simple algorithm is known as the CPython Evaluation Loop ??. The evaluation loop ?? is formulated in the following manner[5]:

Evaluation is computed frame by frame (See **??**), 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 [?]. 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 (**??** and **??**) are essential for the proper computation of any variables that there are in the frame (See **??**). Most instructions manipulate the value stack and the call stack [?]. (andre:) Reworked

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

```
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
    }
}
```

Figure 2.4: Evaluation Loop

Call Stack

The call stack contains call-frames[??] (See ??). This is the main structure of the running program. A function call results into a pushed frame onto the call stack whilst a return call results into a pop of the function frame off of the stack [?]. A visual representation of a call stack is shown in Figure ??. 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 object happens when object-manipulating opcodes are evaluated. A value stack is found in a call-frame implying bijectivity. Any manipulations that are 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.

Python Test Script

```
def foo():
    print("Hello")

def intermediary():
    foo()

def start():
    intermediary()

start()
```

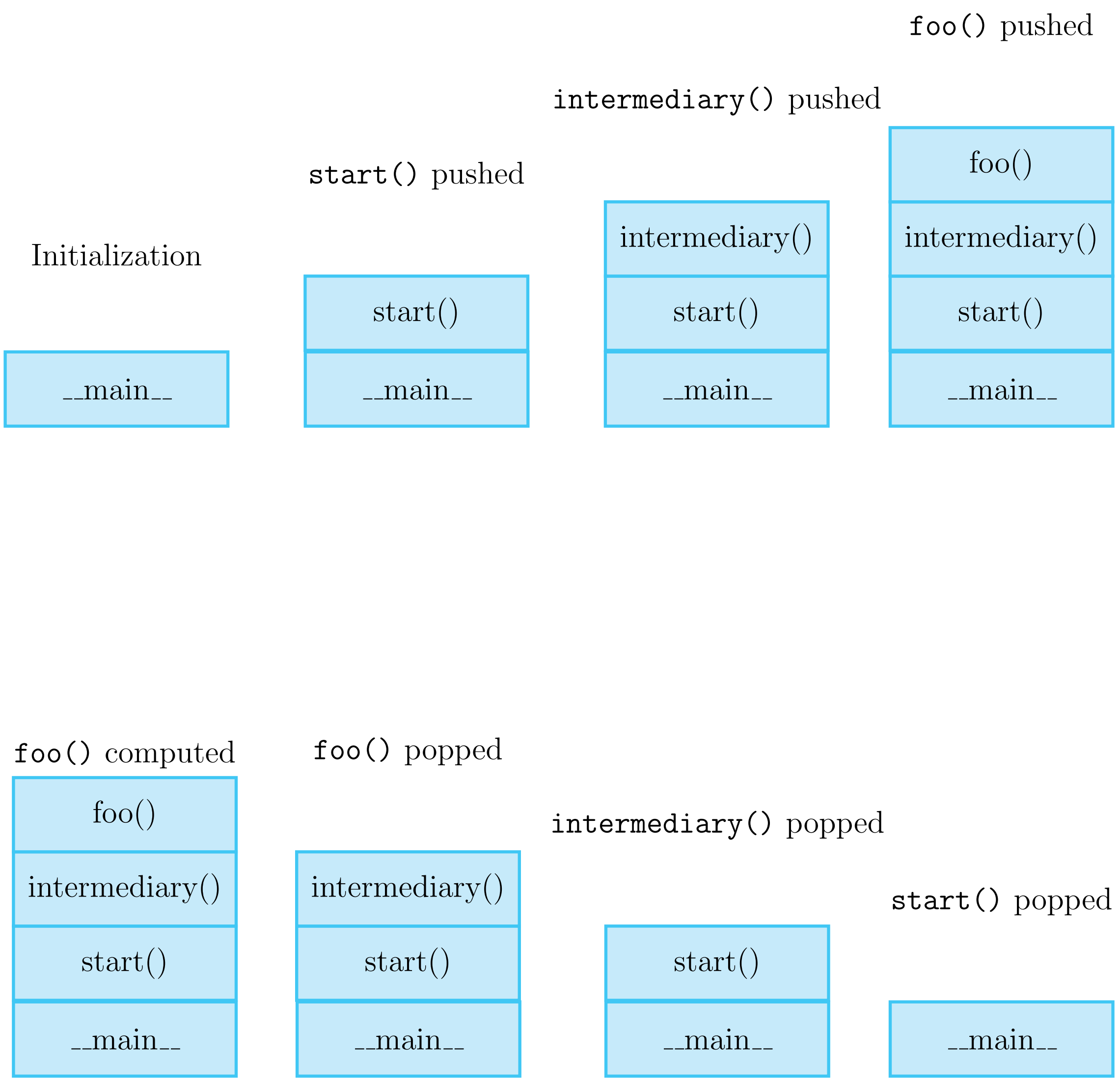

Figure 2.5: Call Stack

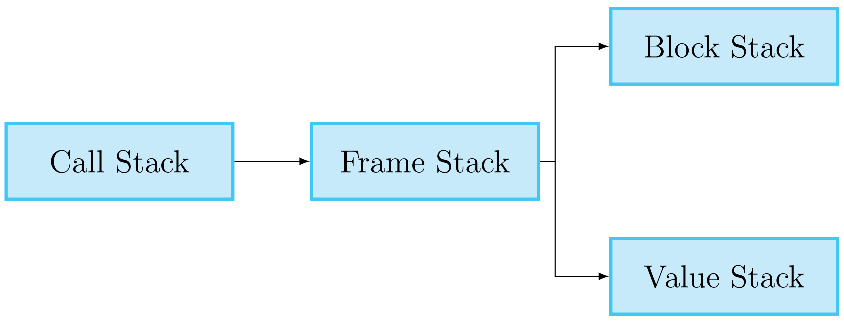

Figure 2.6: Overview of CPython stacks

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 the said structure is exited. The block stack allows the interpreter to keep track of active blocks at any moment

2.2.3 Frames

(andre:) Maybe change???

A frame[6], 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 [**?**, pp.18–19]. Frame objects are tightly coupled with the three main stacks (See **??**) and every frame is linked with another frame. Every frame object has two frame-specific stacks; value stack (See **??**) and the block stack (See **??**). 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 [7]:

[6] Also called a call-frame.

[7] Retrieved from declaration found in *./Include/frameobject.h*

- `f_back`: this is[8] the previous stack frame object (return address).
- `f_code`: this is?? the current code object (See **??**) being executed, in the current frame.
- `f_builtin`: this is?? the builtin symbol table.
- `f_globals`: this is?? the dictionary used to look up global variables.
- `f_locals`: this is?? the symbol table used to look up local variables.
- `f_valuestack`: this is a pointer, which points to the address after the last local variable.
- `f_valuestack`: this holds?? 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 **??**). A frame is pushed onto the stack frame for every function call (as typically, every function has its own unique frame) as shown in Figure **??**.

The stack frame contains a frame pointer which is another register that is set to the current stack frame. Frame pointers resolve the issue that is created when operations (pushes or pops) are computed on the stack hence changing the stack

[8]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.

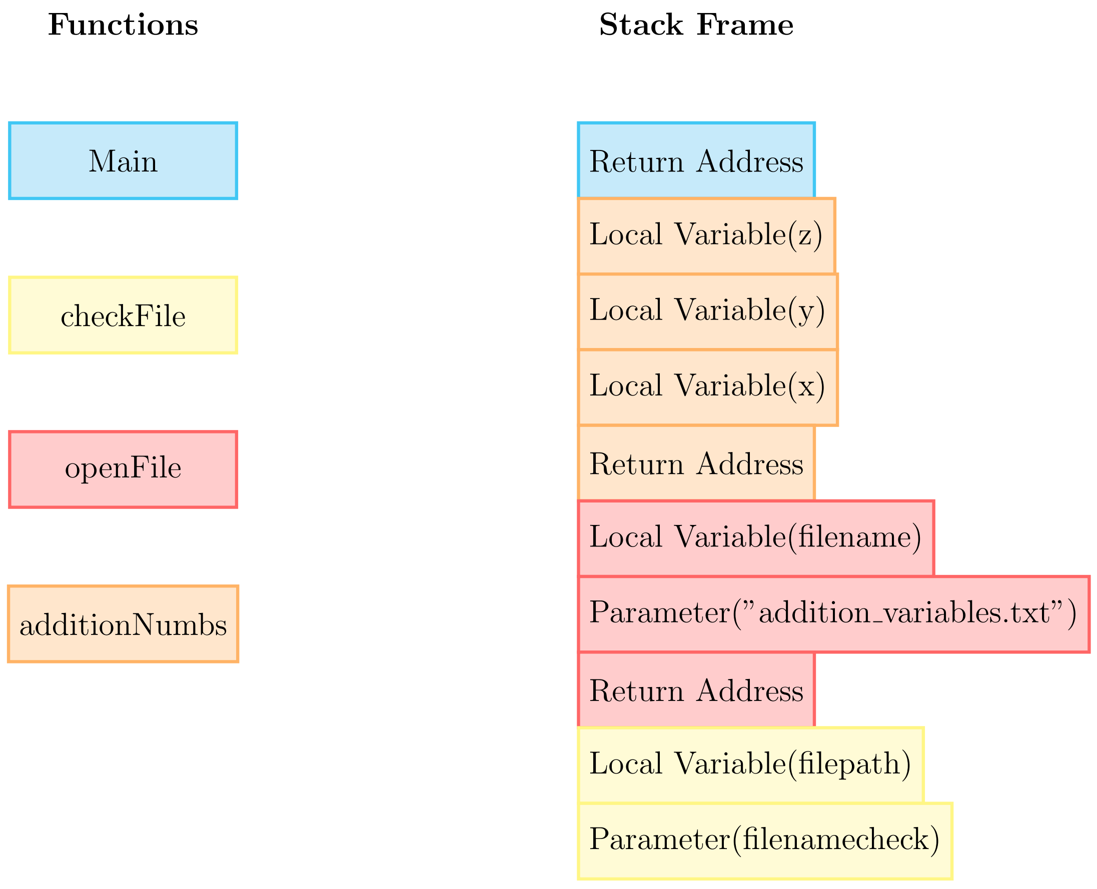

Figure 2.7: Stack frames example

pointer, invalidating any hard-coded offset addresses that are computed statically, prior to run-time [?]. With frame pointers references to the local variables are made as 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 Python code is parsed, it is compiled into a code object to be run on the PVM. A code object contains a list of instructions which directly interact with the CPython VM; hence why it is called a low-level detail. Typically, code objects are of the type `PyCodeObject`[9] and each section of the code object represents a chunk of executable code that has not been bound into a function [?]. The structure of the type of these code objects ?? change throughout different CPython versions, thus there is no set composition [?]. For reference, the source code in Figure **??** produces the following

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

code object:

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

Which follows this convention:

$$codeobject < functionName > at < address >, file < path >, line < firstLineNo. > \tag{2.1}$$

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 [?]. 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 ??. When applying the analysis function `dis.dis`, the disassembled code object takes the following format for every instruction:

$$< lineNumber >< label >< instructionOffset >< opname >< opargs >< var > \tag{2.2}$$

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.

Bytecode

(andre:) Reworked

Bytecode is a form of portable code, that is executed on a virtual machine. The origin of the concept of bytecode was introduced when a generalized way of interpreting complex programming languages was required, so as to simplify information structures that can instead be characterized in their essentials [?].

Python source code

```python
import dis

def addition_numbers(x,y):
    z=x+y
    return z

dis.dis(addition_numbers)
```

Disassembly of Function

```
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
```

Figure 2.8: Disassembly of a Function

This ideology gave birth to portable code[10], 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 [?]. These bytecode operations can be classified in the following manner;

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

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

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

2.2.5 Execution of Code Objects

(andre:) Reworked The evaluation stage firstly makes use of the public API `PyEval_EvalCode()` [?, lines 716–724], which is used for evaluating a code object created at the end of the compilation stages (Figure ??). 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[11] and positional arguments[12].
- The resolution *args[??] and **kwargs[13] 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 [?, pp.2–3].

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 [?].

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).

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

[12]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.

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

- `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 ??, then a new keyword argument dictionary [14] must be created in the form of a `PyDictObject` Similarly, if any positional arguments?? are found they are set as local variables.

The dictionary?? 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 [15] are added to the `*args`?? 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.

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

[14] Denoted as a `kwdict` dictionary.

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

function and forms a vital part of interpretation.

2.3 Analysis

Program analysis is constituted of four main approaches; Control Flow analysis, Abstract Interpretation, Type and Effect Systems, and finally Data Flow Analysis [?, 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 with respect to 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 [?, pp.2–3]. These approaches also have two main methodologies driving them; Statically analysing a program or Dynamically analysing a program.

Program analysis is conduced prior to program input, rendering any analysis undecidable. In complexity theory, this is known as Rice's Theorem [?]. 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 must approximate; producing a *safe* answer [?, 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 the flow of program control may be seen. 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 a number of inclusions out of a program. This system creates relations which can be constituted from three different classes [?, pp.10–13];

- The relation between the values of the function abstraction and their labels.
- The relation between the values of variables and their labels.
- 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 [?, 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 by the means of elementary blocks; connected by edges, describing how control is delegated throughout the program. There are two main methodologies of approaching Data Flow analysis; The Equational Approach, and the Constraint Based approach (as mentioned in Section ??)

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 [?, 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 **??**. 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 (**??**). Abstract Interpretation is a three-step procedure [?, 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.

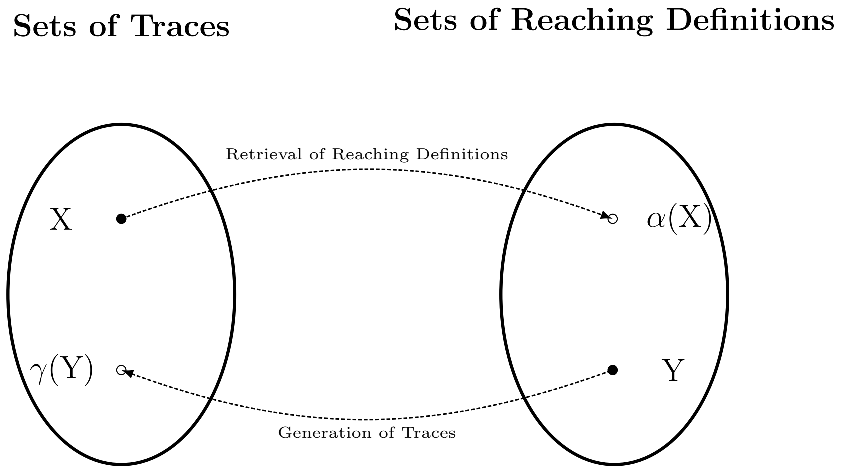
Sets of Traces
Sets of Reaching Definitions
Retrieval of Reaching Definitions
X
α(X)
γ(Y)
Y
Generation of Traces

Figure 2.9: Realization of Galois Connection

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 **??**), 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 [**?**, pp.14–15].

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 **??**. This type of analysis provides a result which is produced efficiently, and relatively precisely.

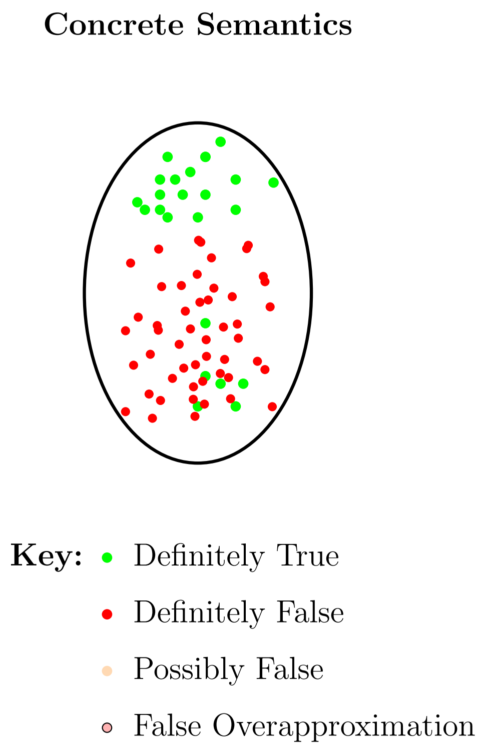

Figure 2.10: Efficiency-Precision trade-off presented by Abstraction Interpretation

2.3.4 Type and Effect Systems

Type and effect systems are the amalgamation of both an Effect System and an Annotated Type System [?, 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 Figure **??**.

Type & Effect System

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

Figure 2.11: SML Code Snippet

3. Methodology

3.1 Abstract Design

PATH is a three-phase, 8-stage bytecode inspection tool (Sections **??** & **??** respectively). The high-level implantation is as follows; PATH iteratively inspects each bytecode instruction produced by the Python Compiler, generating facts as instructions are iterated through. (andre:) Show i include file saving as a phase??

3.1.1 Phase Overview

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

2. The second phase is the iteration phase; BIL(Bytecode Iteration Loop). 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 pulling information from the generated facts in phase 2.

Figure 3.1: Phases of PATH

3.2 Concrete implementation

3.3

3.4 Fact Generation

3.5 IR Generation

4. Evaluation

5. Conclusion

A. Opcode Table

Country List		
Country Name or Area Name	ISO ALPHA 2 Code	ISO ALPHA 3
Afghanistan	AF	AFG
Aland Islands	AX	ALA
Albania	AL	ALB
Algeria	DZ	DZA
American Samoa	AS	ASM
Andorra	AD	AND
Angola	AO	AGO

[?] [?]