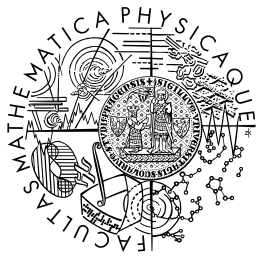

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Martin Zimen

Static analysis of NumPy programs

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: doc. RNDr. Jan Kofroň, Ph.D.

Study programme: Computer science

Study branch: System programming

Prague 2023

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

Dedication. I would like to thank my supervisor for his time, energy and comments, and my parents for bringing me into this world.

Title: Static analysis of NumPy programs

Author: Martin Zimen

Department: Department of Distributed and Dependable Systems

Supervisor: doc. RNDr. Jan Kofroň, Ph.D., Department of Distributed and Dependable Systems

Abstract: NumPy programs can be hard to debug. Due to the dynamic nature of Python, a bug can manifest itself after a long time of run time. This causes the computation to crash, ditching all the progress. Existing static analysis tools can't detect NumPy-specific errors. We propose a solution that uses data-flow analysis combined with symbolic execution to detect `ndarray` shape mismatch errors. With a dynamic set of symbols, our method tracks `ndarray` dimensions and constraints between them throughout the program. It uses an SMT solver to solve the constraints and locate the bug. Our implementation understands core NumPy constructs and detects some shape mismatch errors for 1D and 2D `ndarrays`.

Keywords: NumPy static analysis data-flow analysis symbolic execution

Contents

Chapter 1

Introduction

Python[1] is a very popular programming language for data science. However, for performance reasons, most of the computation isn't done in pure Python. For large data manipulation, a library called NumPy[2] is usually used.

1.1 What is NumPy

NumPy User Guide [1] describes NumPy as follows.

> NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

This makes NumPy the base library upon which other libraries are built. It is essential (among other things) for any application requiring data science such as machine learning. Well-known libraries for machine learning such as TensorFlow,[3] PyTorch,[4] and scikit-learn[5] are built upon NumPy. Not to mention other data science libraries such as Pandas,[6] SciPy,[7] and Matplotlib[8]. It is essential

[1] `https://www.python.org/`
[2] `https://numpy.org/`
[3] `https://www.tensorflow.org/`
[4] `https://pytorch.org/`
[5] `https://scikit-learn.org/`
[6] `https://pandas.pydata.org/`
[7] `https://scipy.org/`
[8] `https://matplotlib.org/`

because it is fast compared to pure Python. This is important for computation-heavy applications.

The speed is accomplished by providing a multidimensional array object called `ndarray` and routines that operate over an `ndarray` (see sections 1 and 5 in NumPy Reference [2]), which are implemented in C. Therefore, the computation-heavy part is not done by Python directly. This improves performance greatly, but it introduces an interface, which can be quite hard to work with.

1.2 NumPy is hard

There are multiple reasons for this.

- Working in NumPy isn't the typical programming we are used to. It is a specific skill, which takes some time to develop and isn't particularly easy.
- Data manipulation code can be quite dense. A lot of work can be done with very little code. If programmers don't adjust and keep the same amount of code per line or function, the program can be quite difficult to read.
- Since speed is crucial, programmers tend to optimize the code. The program is then written in an unexpected way, and it can be quite difficult to deduce what a particular piece of code is supposed to do.
- The programmer has to make sure that NumPy functions are called correctly. Namely, the programmer must make sure that the shapes of `ndarrays` match up. This is particularly hard to do since the shapes of `ndarrays` aren't explicitly stated in the code and may not be obvious at first glance. The programmer is expected to keep track of the shapes by himself. This is hard, and introduces many errors.

1.3 Dynamic analysis debugging

For all the reasons mentioned above, programs which use NumPy can be quite hard to debug. And it gets even worse.

Python is a very dynamic language. Therefore, most of the errors show up at runtime. This is fine if the execution time of the programs is low. However, people usually use NumPy in situations where processing of large amounts of data is needed. Therefore, many NumPy programs run for minutes, hours or more. The error can be located at the end of the program, which would only save the result, plot the data, or aggregate the results. Instead, the program crashes

and all the computation is lost. This will sound very familiar to most NumPy programmers.

This makes typical debugging using dynamic analysis (i.e., running the program) unfeasible or at least ineffective in many applications as the debugging cycle is just too long.

Therefore, the need for other means of debugging arises.

1.4 Small test data

One possible solution would be to prepare a smaller dataset and debug the program on that. This is definitely a possible solution. However, it has its downsides.

Preparation of this data is extra work and can be, in some cases, quite time-consuming.

Even worse, in some cases, this won't reduce the execution time significantly. In some applications, the program can run slowly not because of the large amount of data present. It runs slowly because the number of iterations is so high that, even with small data, the whole program is slow. In these applications (physics simulation, iterative training such as stochastic gradient descent), running the program with smaller data doesn't improve the situation.

1.5 Notebook workflow

One modern solution is to change the workflow completely and utilize other properties of data science.

- Data science program is often run only once and only on one dataset.
- Results are then visualized and discussed.
- Data science code is often very linear.

In this approach, the classic development cycle is abandoned and the development is done with *notebook* tools such as the Jupyter notebook.[9]

Instead of repeatedly editing the code and running it, the work in a notebook is more continuous. Code is separated into a sequence of blocks (this is possible for mostly linear programs). When a block of code is written, it is run and the values are saved. And since the program input isn't expected to change, variable values from this block don't need to be recalculated when they are used by a subsequent block.

[9] `https://jupyter.org/`

This allows the programmer to quickly experiment and debug in a fast fashion. Notebooks also allow the programmer to see the code results in the same place as the code. This also allows for fast inspection of variable content, which aids the programmer in keeping track of `ndarrays`' shapes.

However, this isn't a one-size-fits-all solution. It works well for linear, top-level programs. However, when writing a function or nested code, this approach fails. This is probably the biggest problem with this approach.

This approach also assumes it will be run only once and only on one dataset. Although it is possible to write the code in a notebook and then transfer it to run on other inputs, this won't help with catching bugs that didn't show up with the original dataset.

Also, some people just don't like the notebook approach. It requires a completely different development environment and may conflict with one's preferred and used-to workflow.

1.6 Static analysis

A more traditional approach is to aid the programmer by statically analyzing the code.

For Python, there are many different tools that can help with static analysis. These can be enormously helpful. However, these tools can only detect generic Python errors and not NumPy-specific errors. We elaborate more in chapter 2.

1.7 Scope of this thesis

This is where our bachelor thesis comes in. We try to fill in this gap by creating a static analysis tool that can detect NumPy-specific errors.

We won't be trying to detect any errors which other tools can detect. Our tool doesn't try to compete with them; it tries to complement them. It is expected to be used in combination with other tools. This allows us to focus on NumPy errors better.

This is still quite a big topic as there are many NumPy-specific errors (linear algebra exceptions—when a matrix should be regular but isn't, floating point exceptions, out-of-bounds exceptions and more). Therefore, we will focus only on one type of exception, and that is a *shape mismatch*.

This error occurs when an operation requires some condition on the shapes of the input `ndarrays` and this requirement isn't satisfied. For example, when multiplying two matrices together, it is required that the second dimension of the first matrix must be equal to the first dimension of the second matrix. When

this isn't true, an error occurs. Similarly, when we want to do an arithmetic operation on two matrices, these matrices must have the same shape, or they must be convertible into matrices with compatible shapes.[10]

Therefore, the goal of the thesis is to design and develop a method for analysis of NumPy programs, and to implement this method. It should detect specific shape mismatch errors without the necessity to run the programs, thus making the debug process significantly faster.

1.8 Structure of this thesis

In chapter 2, we discuss existing tools in further detail. In the end, we conclude that we should start working from scratch instead of extending an existing project.

In chapter 3, we explain basic terms and concepts used throughout the thesis. We provide links to resources about NumPy and Python AST, define basic terms for static analysis such as basic block, AST and CFG, and explain the data-flow analysis and related terms.

In section 4.1, we analyze different methods that could be used for the analysis of NumPy programs. In section 4.1.5, we arrive at an algorithm that we decided to use for our problem. In section 4.2 and section 4.3, we shortly discuss what language and UI we should use.

In chapter 5, we further refine the high-level approach from chapter 4. In section 5.1, we focus on details about the algorithm described in section 4.1.5. In section 5.2, we discuss how to use information collected by the algorithm from section 5.1 to find shape mismatch errors.

In chapter 6, we describe our implementation and provide developer documentation for it.

In chapter 7, we discuss how our method and implementation perform on two examples—`linreg` in section 7.1 and `sim` in section 7.2. In section 7.3, we discuss the limitations of our method and implementation.

We conclude with chapter 8.

In appendix A, we provide the user documentation and installation guide. In appendix B, the structure of the ZIP submitted with this thesis can be found. And in appendix C, we provide the unmodified code we derived our test case `linreg` from.

[10] This is called *broadcasting* and is described in section 4.5 of NumPy User Guide [1].

Chapter 2

Related work

Here is a list of projects we considered most relevant.[1]

- PyLint[2]
- PyFlakes[3]
- MyPy[4]
- PyRe[5]
- PySa[6]
- PyT[7]

PyLint and PyFlakes seem to do only very trivial analyses like checking line numbers to detect if a variable is defined. This depth is not enough for our purpose.

MyPy is a well-known and relatively big project which focuses on type analysis. It closely integrates with Python typing. In the last few years, there was an effort to add typing info for NumPy functions so MyPy could detect NumPy-type errors. However, analyzing `ndarray` shapes requires more detailed information than what Python type a variable is. So it would be quite a stretch to make MyPy analyze `ndarray` shapes.

PyT looks like a great project. Unfortunately, it is unmaintained for a few years and reviving it would be too much work.

[1] We found a GitHub site that was very helpful for finding Python static analysis tools. `https://github.com/analysis-tools-dev/static-analysis#python`

[2] `https://pylint.org/`

[3] `https://github.com/PyCQA/pyflakes`

[4] `https://www.mypy-lang.org/`

[5] `https://pyre-check.org/`

[6] `https://pyre-check.org/docs/pysa-basics/`

[7] `https://github.com/python-security/pyt`

PyRe and its subproject PySa look great. Someone even considered extending it in the exact way we wanted (see GitHub issue[8]). It seems that it was even extended to support this kind of analysis (see roadmap[9]). However, this was in 2019, and we didn't find any progress on this. We tried contacting the author of the original issue without any success. However, there would still be problems as PyRe is written in OCaml, which we don't know.

Therefore we decided to start working from scratch. It would probably be easier anyway. Even if a good project existed, getting to know the code base would take a lot of effort.

[8] `https://github.com/facebook/pyre-check/issues/177`

[9] `https://github.com/facebook/pyre-check/blob/main/ROADMAP.md#numerical-stack`

Chapter 3

Definitions and algorithms

3.1 NumPy

Explaining NumPy in detail is way beyond the scope of this thesis. The reader is encouraged to read the chapter *NumPy: the absolute basics for beginners* in NumPy User Guide [1]. Chapter *NumPy fundamentals* provides a more thorough explanation of NumPy's concepts and will be referenced throughout the thesis.

The ultimate source of information about NumPy is the NumPy Reference [2], which will be referenced in cases more specific information is needed.

3.2 Abstract syntax tree (AST)

An abstract syntax tree is a representation of a program. Representing a program in a textual form is helpful for human interaction such as editing. However, for machine interaction with the program, more structural representation is required. An AST provides this structural representation.

An AST depends on the target language. For Python, it is described by the grammar in The Python Language Reference [3]. A more concise version is available in The Python Standard Library Reference [4] in the section about module `ast`. This is the module we used for obtaining an AST for the input program.

An AST consists of *AST nodes*, which represent different language constructs. Any Python program is made out of these constructs. The AST nodes form a tree that represents the structure of the program.

We won't go into details about the Python AST nodes. They are described in the documentation of the `ast` module [4] with many examples. The reader is encouraged to go through the documentation.

```
1  import collections
2
3  def count(text):
4      counter = collections.defaultdict(int)
5      for char in text:
6          counter[char] += 1
7      return counter
8
9  input = input("Enter text: ")
10 freq = count(input)
11 print(freq)
```

Listing 3.1 Example code

The `ast` module displays the produced ASTs in a textual form. This gets unreadable quite quickly. For example, see the program in listing 3.1 and the corresponding textual representation of AST in listing 3.2. This also serves as an example of what an AST of an actual program looks like.

We will need to inspect ASTs of even bigger programs. Moreover, some debugging information from our program can be visualized on the AST as well. Therefore, a better visualization is needed.

We will visualize ASTs more graphically. See figure 3.1 for visualization of the example program from listing 3.1. The program which generates this visualization for any Python program is a part of the debugging toolset we wrote for our thesis.

3.3 Control flow graph (CFG)

A control flow graph provides a more high-level view of a program than an AST does. It represents the control flow through the program.

A program can be split into basic blocks. A *basic block* is a part of code that is always executed sequentially. Therefore, a basic block can't contain any control flow constructs.

A *control flow graph* is a graph over the set of all basic blocks of a program. Nn edge $A \to B$ is part of the CFG if and only if there exists an execution of the program in which a basic block B is executed right after a basic block A. Therefore, every execution of the program follows the CFG edges.

Representing the program via a CFG isn't that common in Python.[1] However, we will use this well-established terminology to talk about static analysis algorithms.

[1] In comparison to languages like C.

```
Module(
    body=[
        Import(
            names=[alias(name='collections')]),
        FunctionDef(
            name='count',
            args=arguments(
                posonlyargs=[],
                args=[arg(arg='text')],
                kwonlyargs=[],
                kw_defaults=[],
                defaults=[]),
            body=[
                Assign(
                    targets=[Name(id='counter', ctx=Store())],
                    value=Call(
                        func=Attribute(
                            value=Name(id='collections', ctx=Load()),
                            attr='defaultdict',
                            ctx=Load()),
                        args=[Name(id='int', ctx=Load())],
                        keywords=[])),
                For(
                    target=Name(id='char', ctx=Store()),
                    iter=Name(id='text', ctx=Load()),
                    body=[
                        AugAssign(
                            target=Subscript(
                                value=Name(id='counter', ctx=Load()),
                                slice=Name(id='char', ctx=Load()),
                                ctx=Store()),
                            op=Add(),
                            value=Constant(value=1))],
                    orelse=[]),
                Return(value=Name(id='counter', ctx=Load()))],
            decorator_list=[]),
        Assign(
            targets=[Name(id='input', ctx=Store())],
            value=Call(
                func=Name(id='input', ctx=Load()),
                args=[Constant(value='Enter text: ')],
                keywords=[])),
        Assign(
            targets=[Name(id='freq', ctx=Store())],
            value=Call(
                func=Name(id='count', ctx=Load()),
                args=[Name(id='input', ctx=Load())],
                keywords=[])),
        Expr(
            value=Call(
                func=Name(id='print', ctx=Load()),
                args=[Name(id='freq', ctx=Load())],
                keywords=[]))],
    type_ignores=[])
```

Listing 3.2 AST of the example code

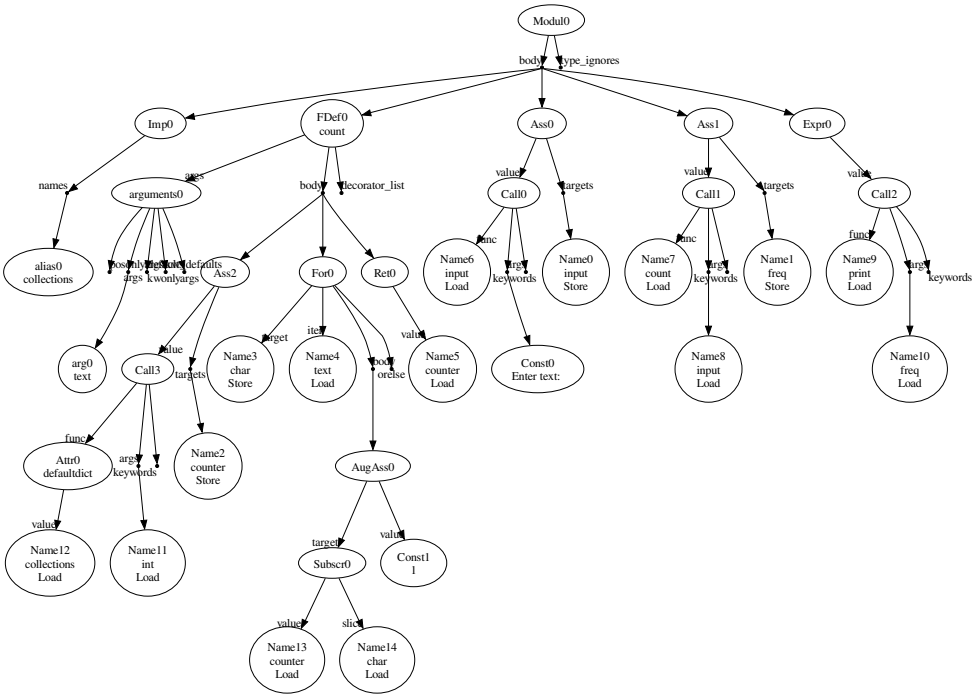

Figure 3.1 Visualization of the AST of the example code

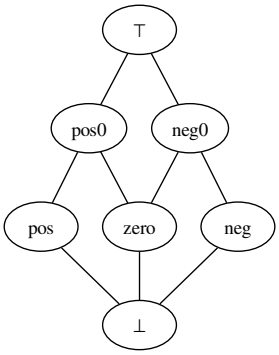

Figure 3.2 An example lattice

3.4 Data-flow analysis

Data-flow analysis[2] is a relaxation[3] method used to get information about possible variable content at CFG nodes. First, we assume we know nothing. Then, the information is improved with the knowledge we have about the basic blocks (e.g., in this basic block, this variable is set to zero), and propagated along the CFG edges. This repeats until no new information is gained.

A lattice is used to store this information. Lattice is a partially ordered set where each pair of elements has a unique supremum and a unique infimum. Each element of the underlying set represents information about the content of a variable (e.g., it is a positive integer). Ordering relation represents the specificity of this information (e.g., a positive integer is more specific information than just an integer).

An example lattice can be seen in figure 3.2. Values `pos`, `pos0`, `neg`, `neg0` and `zero` represent information that the variable content is positive, non-negative, negative, non-positive or zero respectively. Value $\top$ is the *top* of the lattice and represents a set of all integers. That means we know nothing about the variable. Value $\bot$ is the *bottom* of the lattice and represents an empty set of integers. This value is unintuitive but useful. It can be used as a value for uninitialized variables, values that don't have other representation in the lattice or values in an unreachable place (e.g., after an error).

[2]We used the textbook Static Program Analysis [5] to learn about data-flow analysis. However, we describe it a little bit differently.

[3]The relaxation algorithm is a general paradigm for algorithm construction. A suboptimal solution is first assumed and then iteratively improved. See MIT lecture notes [6] for further details.

x	y
B	B
T	T
pos0	pos
neg0	T
pos	pos
zero	pos
neg	neg0

Table 3.1 The update function of `y = x + 1`

Information collected at each CFG node is represented as a mapping from the set of all variables to the lattice. This information could, however, change throughout the basic block which the node represents. One option would be to have two nodes for each basic block representing information before the basic block and after the basic block. We will use just one node per basic block that will represent the information after the basic block.

How to update (i.e., relax) information at a node? If it had only one predecessor, information from the previous block could be taken and updated with changes made to variables in the current basic block. This information is target language dependent and all language constructs must be considered with regard to the used lattice. In our example, an assignment `x = 1` would ignore the old value of `x` and set the new values to `pos`. A basic block consisting of one statement `y = x + 1` would set the value of `y` based on the value of `x`. See table 3.1. The function for updating the values of variables must be constructed for each basic block. We refer to this function as an *update function*.

Now, let's assume a basic block has multiple predecessors. First, the information must be combined from all the predecessors. This can be done separately for each variable. This is the advantage of using lattice to describe the information; the supremum of all the possible values in question can be used. Then, the combined information can be used as if it was the information from an only predecessor of the basic block.

Now, we can run the relaxation algorithm. The easiest (but not very efficient) method is to repeatedly relax all nodes. When we stop inferring new information, we have reached the so-called *fixed point* and our algorithm finishes.

What should we use as the initial value? In nodes with multiple predecessors—we will call them *join* nodes—this initial value shouldn't influence any other values coming from different predecessors. Therefore, we initialize to $\bot$ as $\sup(M) = \sup(M \cup \{\bot\})$ for every set M.

Chapter 4

Analysis

4.1 Algorithm

In this section, we will examine methods for analyzing NumPy programs with regard to finding shape mismatch errors.

4.1.1 Mock run

One possible approach is to replace the `ndarray` with a *mock* and functions that work with `ndarrays` with functions that work with the mocks. The mock doesn't hold any data and just remembers its shape. Then, the mock functions only calculate the new shape for the new mock from the shapes of the input mocks. If the input shapes mismatch, the user is notified. When asked for any actual data, the analysis stops.

This approach is, in a way, similar to running the program with smaller data. The execution time of a single operation with `ndarrays` is reduced. This, however, still leaves the problem that many iterations can slow the program. At least, the need of having to create smaller test data was averted.

On the other hand, it introduces possibly an even bigger problem. Every time a value is needed from an `ndarray`, the analysis has to stop, which limits this analysis technique significantly. Possibly, a separate mock could be used for the values from an `ndarray`, which could be returned. The analysis would stop only after the value is really needed (e.g., when it influences control flow, needs to be printed, or enters a function we cannot or don't want to analyze). This can be helpful when the programmer is doing some simple manipulations with `ndarray` values and saves the values right back into the `ndarray`. However, once this value leaves the `ndarray` or a more complicated manipulation with the value is done, the analysis is stuck.

The main advantage of this approach is its simplicity. Most of the heavy lifting is done by the Python interpreter. The only thing needed is to implement a mock `ndarray`, operations on the mock, and to replace the NumPy library with its mock version at the beginning of the analysis.

It would be definitely interesting to see this implemented. However, we decided to take another approach since this is too weak in our opinion.

4.1.2 Type analysis

Keeping track of different `ndarrays` and determining if a given combination can be used as arguments to a given function feels like a type analysis.

What would the types be? A type could be created for each possible `ndarray` shape and some standard methods for type analysis could be used.

However, this approach reaches its limits quite quickly. What type should a function which creates an `ndarray` (e.g., by loading from a file or converting from some other means of input) return? In many cases, the `shape` of the input array isn't known.

If it was known, the problem would be well suited for work in a notebook. If the shape of the input doesn't change, it is quite rare that there would be a shape mismatch for different inputs. Such a program could be quite well debugged in a notebook.

Overall, having a type for each `ndarray` shape limits the usefulness of such analysis in similar ways the notebook approach is limited.

4.1.3 Parameterized type analysis

It would be useful to have some sort of notion of $n \times m$ matrix type which would be different from $m \times n$ matrix type without actually knowing n or m. This naturally brings in the idea of parameterized types.

We add a set of symbols from which types can be derived. For example, if symbols n and m were available, types $\mathrm{M}(n, m)$, $\mathrm{M}(n, n)$, or $\mathrm{M}(n, 2)$ would get introduced. Type $\mathrm{M}(n, m)$ represents a $n \times m$ matrix.

The input function can then simply return $\mathrm{M}(n, m)$. Functions that modify the `ndarray` shape could preserve those symbols if possible. Transposition could expect $\mathrm{M}(n, m)$ as an argument and return $\mathrm{M}(m, n)$.

With these types, how would the addition of $\mathrm{M}(n, m)$ and $\mathrm{M}(m, n)$ go? One possibility would be to consider all symbols different.[1] Addition would then fail.[2] This approach is problematic.

[1] Different symbols cannot have the same value.

[2] Well, not necessarily. If $n = 1$ matrices could be broadcasted together. This illustrates that broadcasting is tricky and can sometimes result in unexpected behavior in edge cases.

Input function then couldn't return simply $\mathrm{M}(n, m)$. A set of multiple types would need to be returned.

$$\{\mathrm{M}(n, m),\ \mathrm{M}(n, n),\ \ldots\}$$

Construction of this set isn't clear. For example, if we include $\mathrm{M}(n, m)$ and $\mathrm{M}(m, n)$, the transposition of this matrix still includes both elements as well. Therefore, when we add (arithmetically) a matrix to its transposition, no error is detected. When we don't include both, we may miss a valid combination. When multiple matrices with unknown dimensions are introduced, combinations of even more symbols need to be considered.

However, there is an even bigger problem. Let's say a column was added to $\mathrm{M}(n, m)$. One would be tempted to return $\mathrm{M}(n, p)$. However, this leads to quick doom. When removing a column from $\mathrm{M}(n, p)$, a new symbol q would be created to form $\mathrm{M}(n, q)$. Then q should be equal to m. This leads to contradiction as we consider all symbols different. The information that $p = m + 1$ cannot be lost.

One option would be to allow types that can be parameterized by an *expression* over symbols such as $n + 1$ or $n - m$. This blows up the number of possible types even more. Moreover, determining if two expressions are equal is often impossible. When comparing n to $m + 1$, we can never know.

Let's get back to the question of adding $\mathrm{M}(n, m)$ and $\mathrm{M}(m, n)$. This time, we allow symbols to be equal.

The addition implies that $n = m$. We need to somehow keep track of this information. In this example, it could be enough to somehow unify n and m. However, there can be a more complex relationship between n and m. For example, if we analyze broadcasting more thoroughly, we see that

$$n = 1 \vee m = 1 \vee n = m.$$

It seems that the most straightforward solution is to track this information directly. This would also solve the column adding and removing problem since it would be possible to track information that $p = m + 1$ directly.

4.1.4 Symbolic execution

Having a set of symbols and tracking constraints built over this set of symbols resembles symbolic execution. We will now analyze this approach.

Its traditional form as described by King [7] has a big problem. It tries to construct a tree of all possible execution paths. This is alright for mostly linear programs. However, for more complicated programs, the number of possible execution paths explodes. This needs to be solved.

We take inspiration from data-flow analysis. Let's imagine for a while that data-flow analysis worked on a tree of all possible execution paths. The required information would just propagate through the tree towards leaves. If we then combined information from all instances of the same point in code, we would arrive at similar information as data-flow analysis can provide. In this combination process, some information would be lost (such as from which branch a value came) but that information isn't available in data-flow analysis as well.

Data-flow analysis eliminates the explosion problem by basically executing all branches simultaneously. Instead of storing a single value for each execution of the program, it stores the set of all possible values from all executions and propagates them simultaneously.

We will do the same thing with symbolic execution.

4.1.5 Combined approach

In data-flow analysis, we stored and updated the mapping of variables to elements of lattice for every CFG node. Here, furthermore, we need some symbols and constraints over this set of symbols.

Let S be a global[3] set of symbols. Elements of the lattice now contain named references to this set of symbols. For each node, we store a mapping from variables to lattice elements and a logic expression built over S. This information is now the subject of relaxation. We refer to the mapping as just *the mapping* or the *type mapping* in case the term mapping would be ambiguous. We refer to the logic expression as *constraints*.

We will illustrate this approach with the following example from listing 4.1 and lattice from figure 4.1. The `matrix` lattice element contains two named references `shape0` and `shape1`, which correspond to the dimensions of the matrix. The `number` lattice element contains one named reference `value` which corresponds to its value.

Information we would like to have in the exit node could be expressed in two ways.

The first variant uses a set of symbols $S = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$, the mapping in listing 4.2 and the following constraints.

$$z = x + 1$$

The second variant uses a set of symbols $S = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}$, the mapping in listing 4.3 and the following constraints.

$$b = c \wedge a = d \wedge e = b + 1$$

[3]We will explain why we decided to have the set global in section 6.6.

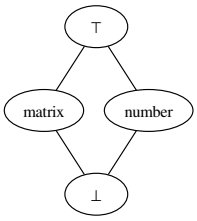

Figure 4.1 An example lattice

```
import numpy as np

# Loads data from file
A = np.loadtxt("input.csv")

# Save the second dimension to variable `c`
n = A.shape[1]

# Create a new matrix full of ones
# which has one more column than `A`
B = np.ones((A.shape[0], n + 1))
```

Listing 4.1 Example code

```
A -> matrix(
        shape0 -> x,
        shape1 -> y)
n -> number(
        value -> y)
B -> matrix(
        shape0 -> x,
        shape1 -> z)
```

Listing 4.2 First variant of the mapping

```
A -> matrix(
        shape0 -> a,
        shape1 -> b)
n -> number(
        value -> c)
B -> matrix(
        shape0 -> d,
        shape1 -> e)
```

Listing 4.3 Second variant of the mapping

Note, that the two variants are equivalent.
This is the high-level approach we decided to go with.

4.2 Implementation language

We wanted to avoid an object-oriented programming language because it doesn't provide enough flexibility for working on a complex problem like static analysis. We would have to bend the paradigm to fit our needs instead of it helping us.

From languages we feel comfortable working with, we considered Haskell for a while because it is a good language for working with programming languages and a functional paradigm seems suitable for this problem. However, we decided to go with Python in the end for multiple reasons.

- Python AST would need to be converted from Python to Haskell.
- Python has support for most external libraries if we needed to use one.[4]
- Analyzing Python in Python is a nice thing. Understanding Python in depth for static analysis makes us more comfortable using it.
- Python doesn't enforce any paradigm and is, in many ways, close to Haskell.

4.3 UI

We want to keep the UI simple and portable. Therefore, we went with a terminal application. The program, which is about to be analyzed, is passed as an argument, and errors are printed to `stdout`. When the relevant node is found in the AST acquired by the `ast` module, the module provides enough tools to print the location of the error.

However, locating the relevant AST node should be enough to extend this tool. It could then work, for example, in IDEs.

[4] In the end, we used Z3. `https://pypi.org/project/z3-solver/`

Chapter 5

Solution

We split our solution into two parts—gathering information with the algorithm described in section 4.1.5 and using this information to find shape mismatch errors.

5.1 Algorithm details

We will now describe this approach in further detail.

5.1.1 Mutability

When we described our approach, we were talking about tracking variables. However, `ndarrays` are objects and variables only hold a reference to these objects. This brings in the problem of aliasing. The properties of an object in one variable could be changed by manipulating the object through another variable.

Because of this, we would have to track all the references to an object. Keeping track of changes made through references would be almost impossible as references can travel through lists, dictionaries or even user-defined objects. And once we lose track of a reference, almost any code can change the properties of the object without us knowing.

Of course, we don't have to do this perfectly. We could track references only so far. Once the reference is passed somewhere we don't understand (saved to a list, a dictionary, an object or passed as an argument to a function we don't analyze), we just give up.

Even this would be really hard as we would need to analyze operations in a much bigger context than we do when only tracking variables.[1]

[1] We still didn't cover every possible problem (such as accessing free variables from nested functions) but this was enough to discourage us from going this way.

Fortunately, the information we want to track is mostly immutable. This means we don't have to distinguish between objects and references. We can treat references as if it was the object directly. So, when we copy a reference, we can think of it as copying the whole object.

Writing to the shape property

The only exception is writing to the `shape` attribute of an `ndarray` directly. This is described in the NumPy Reference [2].

> The shape property is usually used to get the current shape of an array, but may also be used to reshape the array in-place by assigning a tuple of array dimensions to it. [...] Reshaping an array in-place will fail if a copy is required.
>
> Warning: Setting `arr.shape` is discouraged and may be deprecated in the future. Using `ndarray.reshape` is the preferred approach.

However, we decided to just reduce our scope to programs that don't use this feature as it is not worth the hassle to consider `shape` a mutable property. This seemed appropriate since operating with `ndarrays` in this way is discouraged anyway.

5.1.2 Lattice

As we stated in the high-level description of the algorithm, we use a lattice to track information about variables. This information consists of the type information and references to symbols that represent dimensions of `ndarrays`.

Primarily, we want to track `ndarrays`. However, other objects can store the dimension information as well. Shape information is usually represented by a tuple of `ints` and a single dimension is represented by an `int`. These objects aren't stored in variables that often. It usually shows up more as an intermediate result. However, being able to track intermediate results using the lattice will turn out to be useful.

Other intermediate results include tuples of `ndarrays`. These are used as input for stack functions (see section 2.3.2 in NumPy User Guide [1] or section 5.2.6 in NumPy Reference [2]). Slices are used when indexing into an `ndarray`. The dimensions of the resulting `ndarray` are determined by the slice, and should be, therefore, tracked as well. And since slices often come in tuples for indexing, tuples of slices should be represented as well. Mixed tuples of slices and `ints` should be tracked for the same reason.

However, the lattice can track other information that can be useful in the analysis. As we will discuss in section 5.1.6, it is helpful to be able to track the information that a name (i.e., a variable) is a function or a module. Moreover, we will need to track which function it is. Therefore, a lattice element is needed for each function name we know.

Remember, as we discussed in section 5.1.1, we can't track any mutable information. Specifically, we cannot track lists, which are often used for the creation of `ndarrays` and for indexing. This is unfortunate as it would be quite useful.

This leaves us with this list of information to track.

- `ndarray`
- `int`
- tuples of `ints`
- slice
- tuples of slices
- mixed tuples of slices and `ints`
- tuples of `ndarrays`
- module
- function

Note, that this listed information hardly ever comes in combination (e.g., a variable would rarely be an `ndarray` and a function in the same program). Also, having this information wouldn't help us much in our analysis anyway. Therefore, a very simple flat lattice is sufficient. This simplifies many things.

What makes our analysis powerful isn't the lattice anyway. Most of the shape information is kept in the constraints.

The lattice could then look something like this (see figure 5.1).

However, there are some problems. How should we represent tuples of different lengths? And how should we represent mixed tuples?

With heterogeneous tuples, one option is to have a lattice element for all the combinations. This is an infinite amount theoretically. In reality, only small tuples are used. We could represent all the small tuples with their own lattice elements.

Another option would be to introduce a lattice element for "slice or `int`". This breaks the flatness of the lattice but allows us to convert heterogeneous tuples to homogeneous ones.

As for the homogeneous tuples, we would intuitively say we solve it the same way we treat `ndarrays`. Storing shape information about `ndarrays` and storing information about tuples of `ints`—which are used to carry `ndarray` shape information—should be the same thing. We will address this problem in section 5.1.7 when we have details about other components of the algorithm.

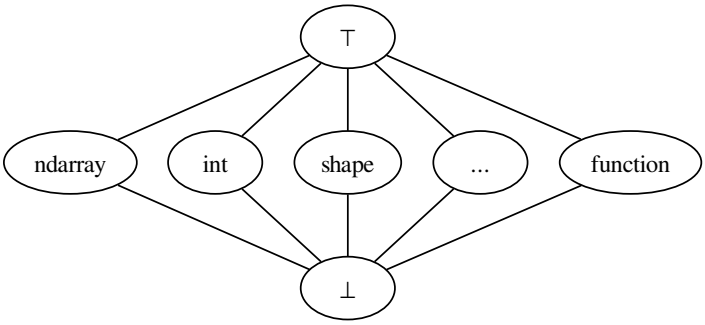

Figure 5.1 Lattice

5.1.3 Relaxation graph

Our algorithm tracks information in multiple points in the program (i.e., nodes) and updates information between them. The selection of these nodes is important for the whole algorithm and will determine how we construct the update function. When we described the data-flow analysis, we were operating on a CFG. We didn't specify what graph we are operating on when we described our algorithm.[2]

Traditional approach

Data-flow analysis is usually done over basic blocks. The effects of a basic block on the mapping are summarized in the update function. When doing an analysis of something simple such as liveness analysis in a language like C, this could be as simple as going through the basic block, finding all assignments and uses of a variable.

However, the analysis needs to be much deeper in our case. We need to detect operations that can change the value of the variable (which is harder in Python than it is in C), and analyze the input of those operations to deduce what the new value will be. The input may depend on the mapping or on the results of other operations. Then, we need to combine this information together.

More fine-grained approach

Instead of summarizing information from the whole basic block, we could make the graph more fine-grained by introducing more nodes. The update function

[2]However, since we just expanded the data-flow analysis from section 3.4, having a CFG in mind as a rough image was reasonable.

between them could then be derived directly instead of combining multiple operations together. This could simplify the implementation enormously by cutting out the middleman.

One option would be to make a node for every statement. This would simplify some things but would still require some aggregation. Python has named expressions[3] and a variable can be changed by a function call when the function is nested, and a variable is declared as `nonlocal`. Also, operations can be nested (like `f(g(x))`). Having only a single operation per update function (edge in the graph) would be really nice. The natural granularity is then the AST node.

With this level of granularity, tracking inputs and intermediate results of operations becomes easy. If a node is an expression, we can just track the value of the expression in terms of a lattice element. Since this is the same information we store in the mapping, we can update the information about the value of the expression with the same relaxation method. We refer to the information about the value of the expression kept and updated in the node as the node's *return value*.[4]

Recipes

We call this new graph a *recipe graph*. We call an update function in the recipe graph a *recipe*. The nodes required to compute a recipe are called *ingredients*. An oriented edge in the recipe graph connects an ingredient with the node it is used to update (it originates in the ingredient).

Although this new terminology isn't strictly necessary, it helps to avoid confusion with the CFG and provides plenty of unique names to use as variable names in an implementation. And since we used them in our implementation, we use them in our thesis as well to avoid confusion.

We will demonstrate the recipe graph with the following example. See source code in listing 5.1 and visualization of AST in figure 5.2. Solid edges in the figure represent AST edges. There will still be some unexplained details, which we will address later.

All AST nodes (e.g., `Call0`, `Attr0` and `Name2`) are also nodes in the recipe graph. Edges of the recipe graph are dashed in the figure. Four recipes were constructed.

[3] Basically, assignments that are expressions instead of statements.

[4] This name was chosen because `return` is a keyword in Python. And since we decided to keep the return value in the mapping in our implementation, this is helpful because it won't interfere with any variable name.

```
# Create an `ndarray` of the same shape as `b`
# and save it to `a`
a = ones(b.shape)
```

Listing 5.1 Example code

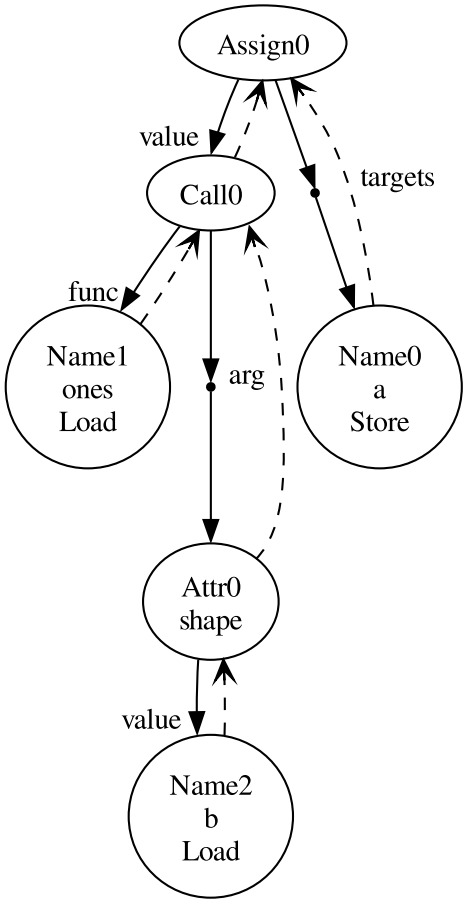

Figure 5.2 AST of the example code

1. The first recipe updates node `Name2` and has a single ingredient `X` (not visualized). This ingredient represents the previous point of execution.[5]
2. The second recipe updates node `Attr0` and has a single ingredient `Name2`.
3. The third recipe updates node `Call0` and has two ingredients `Attr0` and `Name1`.
4. The fourth recipe updates node `Assign0` and has two ingredients `Call0` and `Name0`.

Each node has a mapping, return value and constraints. At the start, the mapping looks like this for every node.

$$\texttt{a} \rightarrow \bot \quad \texttt{b} \rightarrow \bot$$

All return values are set to $\bot$ as well.

1. Let's say the information that `b` is an `ndarray` reaches this part of the recipe graph. To be more precise, it reaches a node `X`. The first recipe updates the mapping of node `Name2`. It propagates the information about `b` and works out that the return value of `Name2` should be the same as `b`.
2. The second recipe updates the mapping of node `Attr0`. It propagates the information about `b` and works out that the return value of `Attr0` should be a tuple of `ints` because the return value of `Name2` is an `ndarray` and the attribute in question is the `shape` attribute.
3. The third recipe updates the mapping of node `Call0`. It propagates the information about `b` and works out that the return value of `Call0` should be an `ndarray` because it somehow[6] got the information about which function it represents. This function creates an `ndarray` of a given shape.
4. The fourth recipe updates the mapping of node `Assign0`. It propagates the information about `b` and works out that `a` should be an `ndarray` because the return value of `Call0` is an `ndarray`.

Recipe construction

The elephant in the room is the construction of these recipes. This will be the subject of the subsequent sections. We will not describe all the details about the construction of every recipe. Instead, we will describe how we handled some bigger problems we encountered. After that, the construction of all the recipes should be straightforward.

[5] We will discuss this more in section 5.1.4.

[6] This will be answered in section 5.1.6.

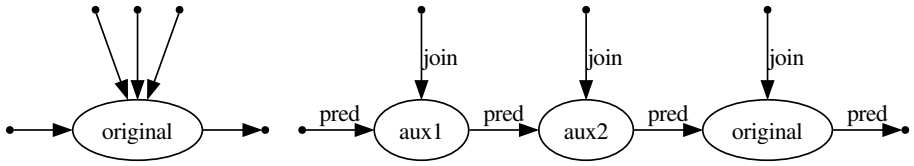

Figure 5.3 Visualization of conversion

5.1.4 Recipes for control flow

A very important task of recipes is to propagate information through the recipe graph in the direction of control flow. Some recipes' sole purpose is to propagate information. And even for some of the more sophisticated recipes, most information is just copied from the node which represents the previous step of the program execution. Therefore, most recipes have an ingredient that serves as the source of context. This ingredient provides the mapping and constraints to be updated with the action done in the next step of execution.

We refer to these ingredients as *control flow ingredients*. We refer to the respective edges as *control flow edges*. We refer to an incoming control flow edge of a node an as *entry edge*. We refer to an outgoing control flow edge of a node an as *exit edge*. Terms entry and exit edges intuitively generalize for other structures such as subtrees.

In most cases, there is only one control flow ingredient. We refer to it as the `pred`[7] ingredient.

In some situations, there are two control flow ingredients. We call them `pred` and `join`. In these situations, control flow joins, and two contexts need to be joined together. A typical example of this situation is the point after an `if` statement—a join node.

We avoid more than two control flow ingredients. This can happen, for example, when there are multiple `break` statements in a `while` loop. We convert this situation to the previous case by adding some auxiliary nodes.

Let there be $n > 2$ control flow ingredients. We add $n - 2$ auxiliary nodes and link them with `pred` edges to form a chain with the original node. One control flow ingredient can be connected as the `pred` ingredient to the first auxiliary node of the chain. This leaves us with $n - 1$ `join` ingredients in other nodes for connecting the rest of the control flow ingredients. See figure 5.3.

[7] This stands for predecessor.

Basic block

We will start simple with a basic block. The order of evaluation in Python is defined very simply in the Python Reference [3].

> **6.16 Evaluation order**
>
> Python evaluates expressions from left to right. Notice that while evaluating an assignment, the right-hand side is evaluated before the left-hand side.
>
> In the following lines, expressions will be evaluated in the arithmetic order of their suffixes:

```
expr1, expr2, expr3, expr4
(expr1, expr2, expr3, expr4)
{expr1: expr2, expr3: expr4}
expr1 + expr2 * (expr3 - expr4)
expr1(expr2, expr3, *expr4, **expr5)
expr3, expr4 = expr1, expr2
```

There are exceptions to this rule. For example, a star argument is always evaluated before keyword arguments (see 6.3.4 in the Python Reference [3]). Unfortunately, we don't know how many exceptions there are. They are scattered through the Python Reference.

This order is the post-order of running DFS over the AST when the children of each node are visited in the order specified by the `_fields` attribute of the node. This even explains the anomaly with the star arguments.

Identification of the `pred` ingredients can therefore be done by running DFS. See figure 5.4 for the result. The entry edge of the basic block is represented by the arrow going from the top to the bottom. The exit edge is represented by the arrow going up from the root of the subtree. We use this representation throughout this section.

`if` statement

First, we will tackle the `elif` clause. This is just syntax sugar for an `if` statement inside the `else` branch of another `if` statement. Therefore, an `if` statement has only three AST child nodes—`test`, `body` and `orelse`.[8]

The control flow goes as follows. First, the `test` subtree is executed. Then, `body` or `orelse` is executed. After that, the control flow is joined and leaves the `if` statement. See figure 5.5 for visualization.

[8]The `else` branch of an `if` statement is called `orelse` in Python AST. This is because `else` is a keyword and would be hard to work with inside Python. We follow the Python terminology.

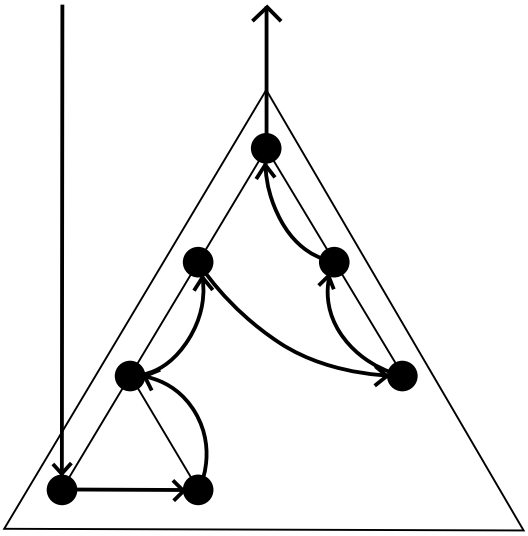

Figure 5.4 Control flow in a basic block

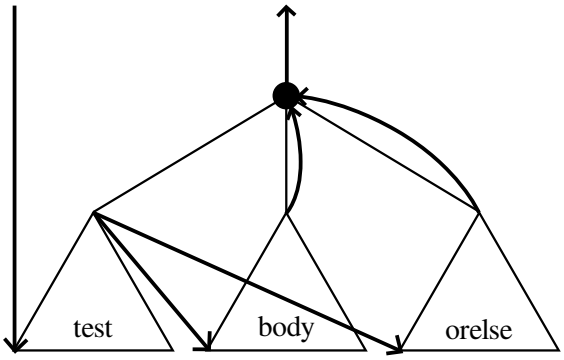

Figure 5.5 Construction of an `if` statement

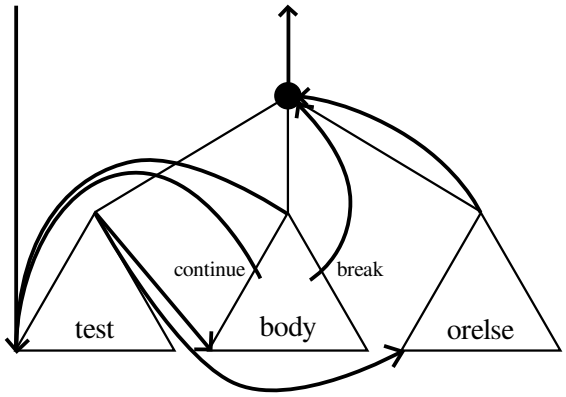

Figure 5.6 Construction of a `while` statement

`while` statement

The control flow in a `while` statement is very similar to the `if` statement. The only difference is that after `body` is executed, the control goes to `test` again. See figure 5.6 for visualization.

`for` statement

This can be done analogously to the `while` statement. The only difference is that `target` is executed before everything else and `iter` is called instead of `test`.

`break` and `continue` statements

Anywhere inside the `while` and `for` statement `body`, there can be a `break` and `continue` statements. This just transfers control to the end of the whole statement or `test` respectively. See figure 5.6 for visualization.

Linking of constructs

These described constructs need to be linked together. This works quite naturally because of the recursive nature of the problem. When encountering a more complicated construct, it can be treated as a simple node with entry and exit edges from the point of the outer structure. The inner construct can be recursively resolved and presented to the outside world through its entry and exit edges.

Other control flow

We ignored all the other control flow constructs. We don't support

- pattern matching,

- `with` statements,
- and `try-except` statements,
- `raise` statements,
- and `return` statement.

The first four constructs are not that common in NumPy programs. The `return` statement would be really nice to implement if we want to analyze functions.

5.1.5 Recipes for working with variables

Setting variables

The question of which constructs change variables (i.e., bind names) is nicely answered in the Python Reference [3].

> **4.2.1 Binding of names**
>
> The following constructs bind names:
>
> - formal parameters to functions,
> - class definitions,
> - function definitions,
> - assignment expressions,
> - targets that are identifiers if occurring in an assignment:
>   - for loop header,
>   - after as in a with statement, except clause, except* clause, or in the as-pattern in structural pattern matching,
>   - in a capture pattern in structural pattern matching
> - import statements.

Of course, since Python is a very dynamic language, there are special ways to change variable content that we won't account for (see listing 5.2).

```
1 vars()["a"] = 42
2 print(a) # Prints 42
```

Listing 5.2 Dynamic variable change

As for the listed constructs, these can be dealt with in the respective recipes. The mapping can be updated with the required lattice element for the target variable name.

Nested scopes

A variable can also change from a nested function when it is marked as `global` or `nonlocal`. We don't support these constructs. These constructs are not that common in NumPy programs.

Using variables

This is also very simple. The recipe can just set the return value of the node to the value found in the mapping for the target variable name.

5.1.6 Recipes for functions

Identifying functions

In Python, functions are treated as first-class objects. They can be therefore identified in the same way `ndarrays` are. A function call is represented with a Call AST node. This node references another expression node. This expression is used as the callable. It would be natural to use the callable node's return value to identify the function. This is why we added the function lattice element. This lattice element internally stores the name of the function. From the perspective of the lattice, there is, technically, a lattice element for each possible name.

NumPy functions are introduced through an `import` statement which does the relevant name binding. Therefore, the respective import recipe should then bind the relevant name to the function lattice element. This works for functions introduced by an `import` statement like `from numpy import transpose`.

The other way to introduce NumPy functions is to import the whole NumPy as a single name via `import numpy as np`. Since a module is, once again, a first-class object, we can just add another lattice element for it as well. This time, only one lattice element is needed since we track only the NumPy module.

Functions from a module are then accessed through the standard attribute construct (e.g., `np.transpose`). The Attribute AST node is an expression therefore its return value can be just set to the needed lattice element which represents the given function.

Representing functions

Let's say we have identified a function recipe—a recipe that actually does some computation. Now, we need to decide how to update the mapping and the return value.

As we mentioned earlier, we don't support `nonlocal` and `global` variables. Therefore, no immutable properties of any variable can change. So, only the return

value of the expression needs to be computed. For this, we need to know what the function does with regard to the information we store in the lattice. Therefore, some sort of description of this function is needed. We call this description a function *stub*.

Python has positional arguments and keyword arguments. Arguments of the call map to the parameters of the function. Parameters are of five types—positional-or-keyword, positional-only, keyword-only, var-positional and var-keyword. When a function call is made, arguments must be mapped to the parameters. This is not trivial. As NumPy uses mostly positional arguments for shape manipulation operations, we decided to completely ignore keyword arguments. This simplifies things tremendously. Arguments and parameters can, then, be thought of simply as a tuple.

An early idea was that each stub would be a Python function that would output the return value based on the mappings and return values of the ingredients. This felt too powerful and too much work as there are lots of functions we need to construct a stub for. This would also make it harder to generate stubs programmatically for user-defined functions.[9]

Another idea was to simply describe a stub with lattice elements and constraints. Arguments would be described with a tuple of lattice elements, the return value (of the function) would be described with a single lattice element and the logic of the function would be described simply with a constraint over the symbols from the arguments and the return value.

When applying this stub, arguments (return values of the argument ingredients) would be mapped to the stub's arguments, symbols in the stub's constraint would be replaced with symbols from arguments, and the return value of the recipe's node would be set to the return value of the stub. It may be needed to generate new symbols for the return value.

The stub's constraints can then be just conjuncted to constraints from the `pred` ingredient.

This idea was then refined in some technical details but is roughly the idea we went with. Also, it is quite universal and can be used in many other situations—operators, indexing, slicing or attributes—not only for the Call AST node.

Polymorphic functions

In some cases, a function or an operation works for multiple input types. We will use addition as an example. When we add two `int`s, we get an `int`. When we

[9] In our implementation, we don't support user-defined functions anyway. But an extension in this direction would be really helpful as we could then analyze the use of libraries, which use NumPy internally, and not only the use of NumPy directly.

add two ndarrays together, we get an ndarray with the shape of the two input ndarrays broadcasted together. The function recipe should support this.

We solve this issue by having multiple stubs for such a function and then just choosing the one where arguments fit on the spot.

In case multiple stubs fit, we could construct the set of possible outputs and then choose the supremum of this set in the lattice. This leaves the issue of correctly setting the constraints. However, the fitting of multiple stubs is usually caused by at least one gotten argument being $\top$. Therefore, we can just solve these situations by returning $\top$ without deeper analysis and we won't lose much information.

Attributes

Attributes are a bit more complicated than polymorphic functions. Polymorphism is not done only over the input types but over the attribute name as well. Therefore, an attribute recipe first checks what the return value of the target is. With this information, we then search for the right stub.

5.1.7 Number of dimensions problem

The number of dimensions is variable in an ndarray and needs to be represented. The same problem arises for representing tuples of ints or tuples of slices—their length needs to be represented.

Number of dimensions as a symbol

The first option would be to have a single ndarray lattice element and represent the number of dimensions with a symbol. Other symbols would then represent the shape information. This is possible, but there are some problems.

Number of symbols We would like to have the set of symbols for each lattice element static. However, this approach requires a dynamic number of symbols. This could be solved by having a set of symbols large enough and then just using some of them. This brings in the question of how to set this "large enough" number of symbols.

We tried to find if NumPy has a maximum number of dimensions. NumPy has the NPY_MAXDIMS constant which specifies this. We didn't find any particular value in the NumPy Reference [2]. However, a quick experiment (see listing 5.3) shows this value is set to 32 in our version of NumPy.

So, it seems it would be safe to set this value to 32. However, a smaller value should be sufficient for most applications.

```
>>> np.ones([2 for _ in range(100)])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.11/site-packages/numpy/core/
      numeric.py", line 191, in ones
    a = empty(shape, dtype, order)
        ^^^^^^^^^^^^^^^^^^^^^^^^^^
ValueError: maximum supported dimension for an ndarray
    is 32, found 100
```

Listing 5.3 Maximum dimension

Stubs What would the stubs look like with this approach? We will demonstrate this on the `np.transpose` function. Let `x.y` refer to the symbol `y` of an argument or a return value—the lattice element `x`.

$$\begin{aligned}&\text{arg0.ndim} = \text{rv.ndim}\\ \wedge\,&(\text{arg0.ndim} = 1 \Rightarrow \text{arg0.shape0} = \text{rv.shape0})\\ \wedge\,&(\text{arg0.ndim} = 2 \Rightarrow (\text{arg0.shape0} = \text{rv.shape1} \wedge \text{arg0.shape1} = \text{rv.shape0}))\\ \wedge\,&\ldots\end{aligned}$$

Basically, this is a case analysis anyway. Not particularly nice.

To make this approach with symbols nicer, our logic would need to support indirect referencing of symbols. This would make the logic really powerful and it would be really hard to find any tools for working with it.

It seems we are stuck with case analysis. However, case analysis can be done at the lattice level.

Number of dimensions as lattice information

This would mean introducing a new lattice element for each possible number of dimensions of an `ndarray`. Case analysis would then be done using polymorphic functions. This is the approach we decided to go with as it was nicer to implement.

Likewise, we introduced a new lattice element for every length of a tuple. For mixed tuples of `int`s and slices, we introduced a new lattice element for every combination.

It could be also helpful to have a *universal `ndarray`*[10] lattice element representing an `ndarray` of an unknown number of dimensions. This breaks the flatness of our lattice as a universal `ndarray` is more specific than $\top$ and less

[10] A better name would probably be a *general* `ndarray` but we use this term for something else later.

specific than any lattice element representing an `ndarray` of a specific number of dimensions.

However, this problem can be solved by having the universal `ndarray` at the same level of specificity as `ndarrays` with specific dimensions. This would mean that some situations would be resolved inaccurately but soundly. The supremum of any pair of `ndarrays` (specific or universal) would be resolved as $\top$.

We decided to ignore the universal `ndarray` completely in our implementation.

5.1.8 Recipes for joins

So far, every recipe had only one control flow ingredient. However, contexts from multiple edges need to join sometimes. We already reduced the problem to having only two entry edges in section 5.1.4. These edges come from the `pred` and `join` ingredients. We will now describe how to join the contexts.

Join happens at the statement level. Therefore, no return values are present. Only the mapping and the constraints need to be joined.

Constraints

Joining constraints is simple. The resulting constraint should simply be the disjunction of the two original constraints. This is intuitive since we came from one branch *or* the other.

Mapping

Every variable of the mapping can be joined independently.

- In the simplest case, both variables map to the same lattice element, and the references of these lattice elements point to the same symbols. In this case, the resulting lattice element can just be copied from one of the sources.
- In the other extreme, the variables map to different lattice elements. Since the lattice is flat, only four combinations are possible.
  - Combination $\top$ and $\bot$ is resolved as $\top$.
  - Combination $\top$ and X is resolved as $\top$.
  - Combination $\bot$ and X is resolved as X.
  - Combination X and Y is resolved as $\top$.
- In the last case, the two variables map to the same lattice element, but different symbols are referenced. The resulting lattice element should be

the same as well. However, it is unclear what the new referenced symbols should be. This can't be expressed through the mapping alone.

We used the following approach. Each mismatched reference can be joined independently. Let x and y be the two original symbols for the mismatching reference. We introduce a new symbol z and set the resulting reference to it. We add a new constraint.

$$z = x \vee z = y$$

This constraint can be conjuncted to the joined constraints.

5.1.9 Calling the recipes

In the previous sections, we described the construction of recipes. These are used in the relaxation algorithm to update the information about nodes in the recipe graph. In this section, we describe more thoroughly how these recipes are called and tackle some problems related to relaxation.

Relaxation

Usually, in the relaxation algorithm, nodes that need to be relaxed are kept in a data structure. When a node V is relaxed, all the nodes that depend on the information from V are added to the structure. The algorithm then just repeatedly takes a node from the structure and relaxes it.

However, a much simpler approach is possible when we notice that we can relax a node that doesn't need relaxation. Then, we can just sequentially relax all nodes and repeat. When we do this in the pred-order of the DFS, the relaxation happens in the direction of control flow and is quite effective. There is another advantage—the algorithm doesn't get trapped in an infinite loop. Only in some cases, the information won't be fully relaxed.

We decided to implement the simpler approach. However, we need to detect the fixed point manually. Therefore, after each pass, we check if some information in the mapping or constraints has changed.

Fixed point detection

Detecting a fixed point in the mapping is simple. When the mapping hasn't changed and all the lattice elements reference the same symbols, we have reached a fixed point.

Originally, we were hoping we could detect a fixed point in constraints just by syntactically comparing them. Unfortunately, in the end, we had to compare them semantically.

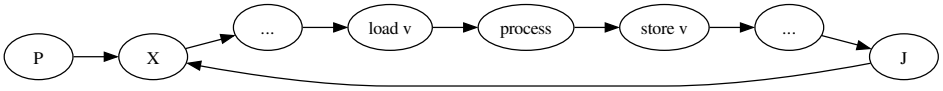

Figure 5.7 Infinite loop situation

To be more precise, we implemented semantic checks, so a new syntactic version of the same constraints isn't generated. In the join recipes, before we update the last constraints with the new constraints, we check if they are equivalent. If they are, we don't change them.

This was done mainly to stop accumulating constraints that looked like this.

$$\varphi \vee \varphi \vee \varphi \vee \ldots$$

Creating new symbols

In the function and join recipes, we sometimes create a new symbol. Since we are relaxing nodes that don't need relaxation, we need to make sure that we introduce new symbols only when it is needed. Otherwise, all nodes would be flooded every pass with new symbols before there was any chance to arrive at some conclusion with the last wave of symbols.

The introduction of new symbols is needed when the referenced symbols, which the new symbols depend on, change. In the function recipes, this is when the arguments change; in the join recipes, this is when the mismatched symbols change.

Infinite loop problem

The creation of new symbols could, however, lead to an infinite loop.

Assume, there is a cycle consisting of control flow edges. See figure 5.7. There is a join recipe in the cycle at a node X. We denote P the `pred` ingredient of X and J the `join` ingredient of X. The node J is part of the cycle; the node P is not. Also, there is a series of recipes that load a variable `v`, create new symbols, and store the resulting lattice element back to `v`. This could be something simple as `v = np.transpose(v)`.

Assume, the node P has a mapping in which `v` is stable in time. In the first application of the join recipe, the mapping in X just propagates the information from P. The information then propagates through the cycle and symbols are created. The mapping in J now contains a new set of symbols for `v`. The following will repeat forever. The join recipe constructs a new set of symbols for `v`. This

```
# Start with a 3x2 matrix
a = np.ones((3, 2))

while ...:
    # Append a column to `a`
    a = np.hstack((a, np.ones((3, 1))))
```

Listing 5.4 Column adding loop

new set of symbols forces a regeneration of symbols in the function recipe. It then propagates through the cycle and J now contains a new set of symbols for v.

This isn't necessarily a bad thing. Consider the example in listing 5.4, where we gradually build a matrix column by column.

In every iteration of the loop in our analysis, new information is acquired about the possible number of columns. Namely, in each iteration, we find out that the matrix a can have the number of columns one bigger than in the previous iteration. However, it takes an infinite amount of iteration to arrive at the conclusion that a can have any number of columns.[11]

At some point, the analysis has to stop.

Widening

When we stop the analysis, we still need to update the information, so it is sound. We call this *widening*.

In the example with adding columns, we cannot simply state that the possible number of columns is in the range 2–something. This information is accumulated in the join recipe and is created by the disjunction of constraints from the control flow ingredients.

Some approximation is needed. Instead of producing a new symbol that is set to be equal to one symbol or another symbol, a new symbol is created which can be anything—it isn't part of any constraint. In other words, we just omit to add constraints when combining mismatching symbols. We also stop creating new symbols even if the mismatched symbols changed. This way, we have soundly concluded that the value can be anything. It may be inaccurate, but it is sound.

Every cycle contains at least one join recipe. And since all join recipes will produce stable information from now on, a fixed point will eventually be reached.

[11] This is similar to a data-flow analysis with a lattice of an infinite height.

Major and minor passes

If we do widening too early, some information that could reach a fixed point may be resolved pessimistically. On the other hand, every pass introduces new symbols which creates bigger constraints that take longer to solve.

To address these problems, we introduced *major and minor passes*. In a minor pass, join recipes are not used for relaxation. This allows us to still propagate information without creating a lot of new symbols. We can keep the number of major passes low, creating a reasonable amount of constraints, while also sufficiently propagating information throughout the recipe graph.

5.2 Analysis of the collected information

In the previous chapters, we described how we collect information about `ndarray` dimensions in the analyzed program. For every point in the program, we have information about what is stored in each variable. This is expressed with a mapping from the variable names to lattice elements, and with constraints over a set of symbols. These symbols are referenced from lattice elements.

In this section, we will use this information to find shape mismatch errors in the input program.

5.2.1 Validation vs. bug hunting

Finding all errors and only errors in an arbitrary program is an undecidable problem. However, this doesn't mean we can't detect anything. We can run an analysis, but there will be situations in which we won't be sure if there is an error. In those situations, we have two options.

Either, we can report this situation as a possible error. We call this approach *validation*. In validation, all errors from a given class of errors, shape mismatch errors in our case, are found. On the other hand, some errors will be false positives. The user can then decide how to handle the reported errors. He can ignore them, or he can provide additional information to the validation program to convince it that the program is correct. There is a big advantage to this approach. When the validation tool doesn't report any error, the user can be sure that the program won't crash with a shape mismatch error. This is helpful when the program is supposed to run for several hours.

Or, we can report errors only when we are completely sure there is one. We call this approach *bug hunting*. In this case, our program doesn't guarantee that there aren't any errors when it successfully finishes. However, when it reports an error, the user can be sure that the error is genuine.

Both approaches have their advantages and disadvantages, and it is ultimately up to the user what he prefers. We prefer the bug-hunting approach. The tool can be just part of a pipeline when we debug, and we know about it only when there is a problem.

There is another reason why we decided to go with bug hunting. When we were starting, we didn't have much confidence in our program. Therefore, we assumed there will be a lot of situations where our program won't be sure. And being bothered with too many false positives would render our program useless. On the other hand, a silent tool, which is run automatically before every run of the program, and only sometimes reports an error, can be useful even if it is relatively weak.

Looking back at this issue, it may have been better to go with the validation approach. There is often a sneaky degenerated input matrix[12] for which the program works. And because of this input, our program can't be sure if there is an error. Therefore, in those uncertain situations, it would probably be beneficial to report the error. Unfortunately, we discovered those degenerated input matrices too late.

5.2.2 Bug hunting as a decision problem

With the information we collected, it is actually quite simple to decide if there is a shape mismatch in the program. We can just simply take the exit node[13] of the program and determine if the constraints are satisfiable. The constraints are in a form that could be solved by an SMT solver.

This is great, but it doesn't provide much information to the user.

5.2.3 Locating an error

If we want to provide any useful information to the user, we need to find the error.

Two sets of constraints approach

One option would be to have two sets of constraints for every stub. The first set would be always satisfiable; it would only describe the return value. The other set would describe constraints that must be satisfied for the operation not

[12] As an example, try to find an `ndarray` `m` such that `m.ndim` = 2, `m.shape[0]` ≠ `m.shape[1]` and `m` + `m.T` doesn't produce an error.

[13] An exit node is the "last" node of the program. More formally, it is the origin of the exit edge of the whole AST. Analogously, an entry node is the destination of the entry edge of the whole AST. There is always only one entry and only one exit node.

to fail. When running the relaxation algorithm, only the first set would be used. Then, when the relaxation algorithm finished, the satisfiability of the second set would be tested in each node, given the constraints collected from running the algorithm.

Border node approach

However, the location of the error can be found even if the constraints aren't separated into two sets. For brevity, we call a node whose constraints are satisfiable `sat`. We call it `unsat` otherwise.

When the exit node is `unsat`, every control flow path from the entry node to the exit node must contain an `unsat` node whose predecessor is `sat`, and every successor on the path is `unsat`. We call this node a *border* node.

Note that a border node cannot be a join node. When a join node is `unsat`, all its control flow ingredients must be `unsat`. If at least one set of constraints is satisfiable, then the disjunction, which contains the satisfiable set of constraints as a disjunct, is satisfiable. Constraints conjuncted for the mismatching symbols are satisfiable trivially.

In all other recipes, we only conjunct to the constraints. Most conjuncts are trivially satisfiable. Only function recipes are an exception. We can remove the part of constraints that only describes the return value; it is trivially satisfiable. This leaves us with the part which checks the operation doesn't fail. Therefore, a border node is always a function recipe node that represents an operation that fails.

Therefore, to locate an error, we can just locate a border node. This can be done simply by following the `pred` ingredients from the exit node. If multiple border nodes exist, we just pick the one found by this approach.

This approach has another advantage. Consider the example in listing 5.5.

```
1 A = np.ones((..., 2))
2 B = np.ones((..., x))
3 C = np.ones((..., 3))
4
5 np.vstack((A, B))
6 np.vstack((B, C))
```

Listing 5.5 Combined error

In this example, there isn't a single function which is wrong. However, an error will definitely occur since the number of columns of `B` cannot be two and three at the same time.

This approach detects errors like this. From the set of operations that contains an error, it will report the latest one. In other words, it uses the information "if

the previous operation didn't fail, constraints for the input must be satisfied".

We decided to go with this approach as it is the more powerful one.

5.2.4 General matrices

When testing, sometimes, an error we expected our program to find remained unnoticed. After long hours of debugging, we realized that there actually isn't any error because there is an unexpected input for which the program works.

For example, with a square matrix, many errors remain undiscovered. For example

- Using the `.shape[0]` attribute instead of `.shape[1]` and vice versa.
- Forgetting to transpose the matrix.
- Swapping the slices when indexing.
- Multiplying the matrices in the wrong order.

Similarly, when some dimensions are equal to one, unexpected broadcasts can happen.[14]

In these cases, the constraints are satisfiable, and no error is reported. However, this hides a lot of genuine errors. It would be really helpful to inform our program that we want to ignore these degenerated cases.

This is an unfortunate problem because it is impossible to reliably determine if the degenerated case is intended or not. This isn't information that is expressed in the source code. This is something the programmer knows.

Detecting general matrices

Ideally, the user would tell us which matrices can be degenerated. But we don't like this. We wanted our program to be sitting quietly in the background and, once upon a time, report an error that the programmer didn't notice.

Ultimately, this is a UI problem we don't really want to get into.

Our best bet is to use a heuristic. When the matrix is loaded from a file, it probably isn't intended to be degenerated. In other words, matrices that are returned by a function that reads input from a file are tagged as general matrices. The list of these functions was made by hand.

Using general matrices information

The information that a matrix is general can be used when analyzing the constraints. Most straightforwardly, a new constraint can be added that represents

[14] Consider this a hint to the problem stated in the footnote in section 5.2.1.

the information that these matrices are general. Degenerated matrices are then not considered by the SMT solver when evaluating satisfiability.

Simple constraints What should this constraint look like? We went with two conditions.

1. No dimension of a general matrix should be equal to one.
2. A general matrix shouldn't be square.

This can be accomplished with this set of constraints for a 2D matrix `m`.

$$\texttt{m.shape0} \neq 1 \land \texttt{m.shape1} \neq 1 \land \texttt{m.shape0} \neq \texttt{m.shape1}$$

With this simple approach, the information about general matrices is preserved as long as the matrix doesn't change shape. If such a derived matrix needed to be square, constraints would force the original matrix to be square leading to a contradiction. Therefore, all the constraints would be unsatisfiable.

However, once a shape is modified (e.g., by adding a column), this generality information is lost. This is quite common, for example, in machine learning where a column of biases is often added.

More robust definition of generality Therefore, we would like to express the idea of a general matrix a bit more robustly than with the previously mentioned conditions.

We would like to express that one dimension of the matrix shouldn't be determined by the other. We didn't find a way how to express this directly using constraints. Therefore, we went with a more complicated approach.

When checking if constraints are satisfiable, one dimension of a general matrix is fixed to an arbitrary value. If this makes the constraints unsatisfiable, then the matrix isn't considered general.[15] Then, a model of the constraints is acquired—an evaluation of the symbols. Then, a new constraint is added. It states that the other dimension cannot be equal to the value it was assigned in the model. The new set of constraints is checked again for the satisfiability of these new constraints. If it became unsatisfiable, the matrix isn't general.

In other words, for an arbitrary value of one dimension, there should be at least two possible values for the other dimension.

This works reasonably well in practice when used in combination with the first previously mentioned condition ($\texttt{shapeX} \neq 1$). When columns or rows are added, and the derived matrix would be a square, the original matrix is forced into a non-general case where one dimension is fully determined by the other. This approach detects these scenarios.

[15] This was added later as a new condition for generality.

Non-general cases Of course, there are many situations when the program isn't expected to work on general input. Therefore, this feature should be optional and the user should be able to turn it on and off.

Multiple input matrices

A similar situation was observed for multiple input matrices. An operation between matrices from different inputs was expected to fail. However, an input combination was found that works (e.g., the number of rows in one matrix is one bigger than the number of columns in the second matrix).

It would be nice to introduce a similar heuristic for this case as well. However, expressing this relationship between multiple matrices is significantly harder. In many cases, the input is expected to match in one dimension (e.g., number of columns). We couldn't figure out one simple heuristic which could be turned on and off. Multiple possible scenarios would have to be introduced.

We decided to ignore this in our implementation.

Chapter 6

Documentation

6.1 Overview

Each function is processed separately. Inter-procedural analysis would be desirable but isn't part of this thesis. However, some information is still propagated from the module scope to functions. If a variable content is consistent throughout the module scope and is a function or a module, this information is propagated to functions.[1]

See figure 6.1 to get an overview of what is done for each function.

In our implementation, *the algorithm* is separated into two parts. Propagation of type information and constraints is done separately.

First, the AST is acquired from the `ast` module and split into functions. Then, for each function separately, preparation for the main algorithm begins. Ingredients are identified with an algorithm based on DFS. Then, recipes are constructed for each node. Because the ingredients were already identified, recipe construction can be done separately for each node. Then, the main algorithm begins.

First, only *the mapping* is updated. When the mapping reaches a *fixed point*, or the maximum number of passes (specified with the `MAX_TYPE_PASSES` argument), we continue to the next phase.

Then, only *constraints* are updated. This is done with *major* and *minor* passes. After each major pass, the minor passes are repeated until a fixed point or the maximum number of passes (specified with the `MAX_MINOR_PASSES` argument) is reached. Major passes are repeated until a fixed point or the maximum number of passes (specified with the `MAX_MAJOR_PASSES` argument) is reached. Then, we do the *widening*. This is, again, followed by a series of minor passes (this time, it propagates constraints through the join nodes but doesn't create new symbols).

[1] To be more precise, this information is set to the mapping in the entry node.

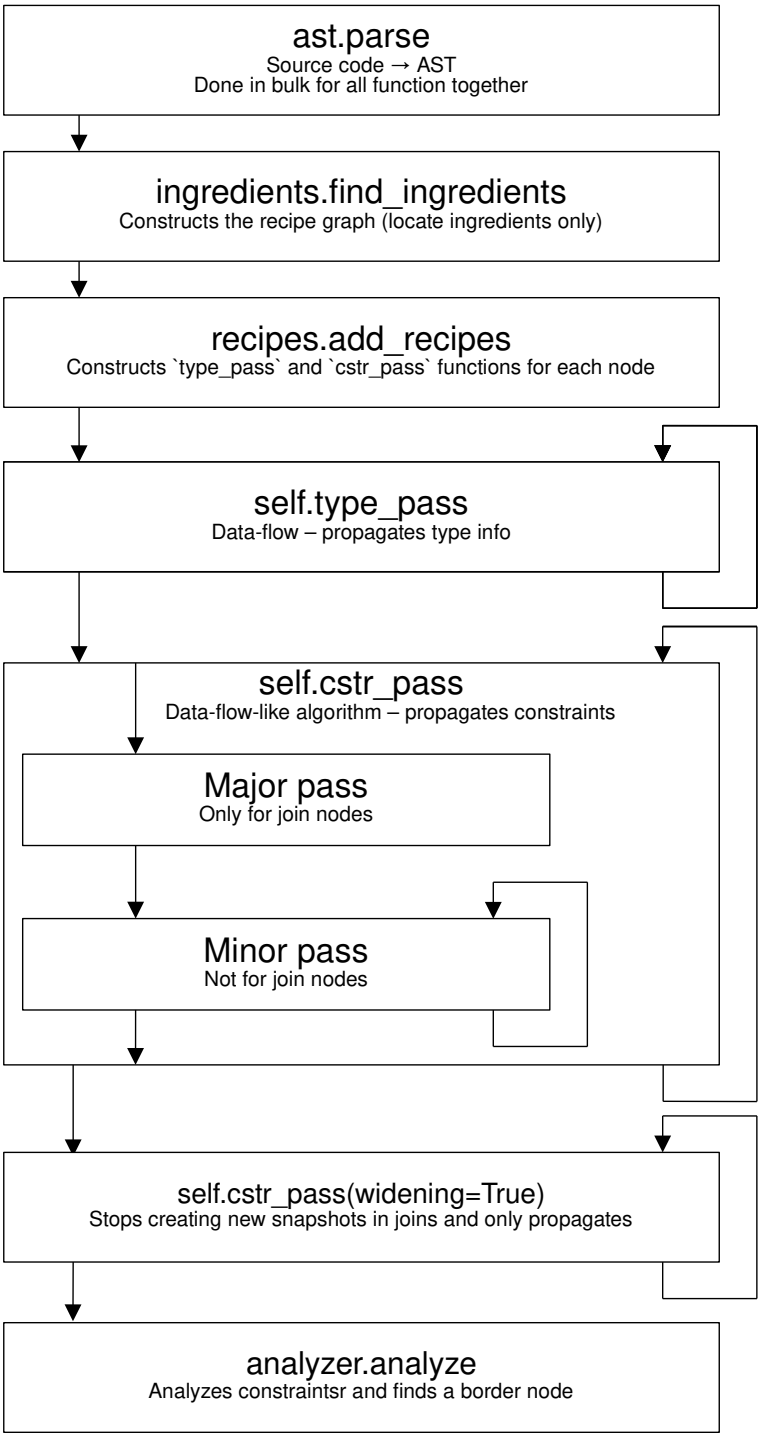

Figure 6.1 Process diagram

In the end, collected information is analyzed using the Z3 solver, and errors are reported.

6.2 Nodes

A *node* is the basic building block of our analysis. Together with *recipes*, they form a *recipe graph*.

Most nodes are monkey-patched AST nodes generated by the `ast` module.[2] If not, the node is an *auxiliary (aux) node*. Every aux node has a parent node. The parent node cannot be an aux node. This allows me to still work with the nodes in a tree-like way. The AST with aux nodes is referred to as *the tree*.

The vast majority of data is saved directly in the nodes. This means that the tree is the main state of the program. For exceptions, see "Reset global state" in `main.py`. The data stored in a node is visualized in figure 6.2.

Dictionary `vars` contains the mapping as a dictionary. The keys for this dictionary are obtained from the `symtable` module. The *return value* of a node is stored in `vars` as well under the key `return`.[3]

Variable `cstr` contains a logic expression that expresses the *constraints*.

6.3 Lattice elements

Lattice elements (abbreviated as `latels` in code) form the *lattice*. They are custom objects, which inherit from the `Latel` class. Most lattice elements are then simply defined with a list of symbols they contain.

Lattice elements are shared across nodes.

6.3.1 Lattice elements types

An `ndarray` is represented by the `Matrix` and the `Array` classes. Each class represents a different possible number of dimensions.

Function lattice elements are represented by a single `Function` class which contains the name of the function as a string.

Tuples are represented by multiple classes. Each represents a combination of elementary lattice elements (e.g., tuple `(int, slice)` has its own class). Not all combinations were implemented. While nested tuples could be theoretically

[2]We tried to monkey-patch the whole module, so the nodes don't have to be monkey-patched individually. Unfortunately, this is not possible because the `ast` module is filled with internal Python magic, and the creation of the nodes seems to be done by the Python interpreter itself.

[3]The name `return` is a keyword in Python a won't interfere with variable names.

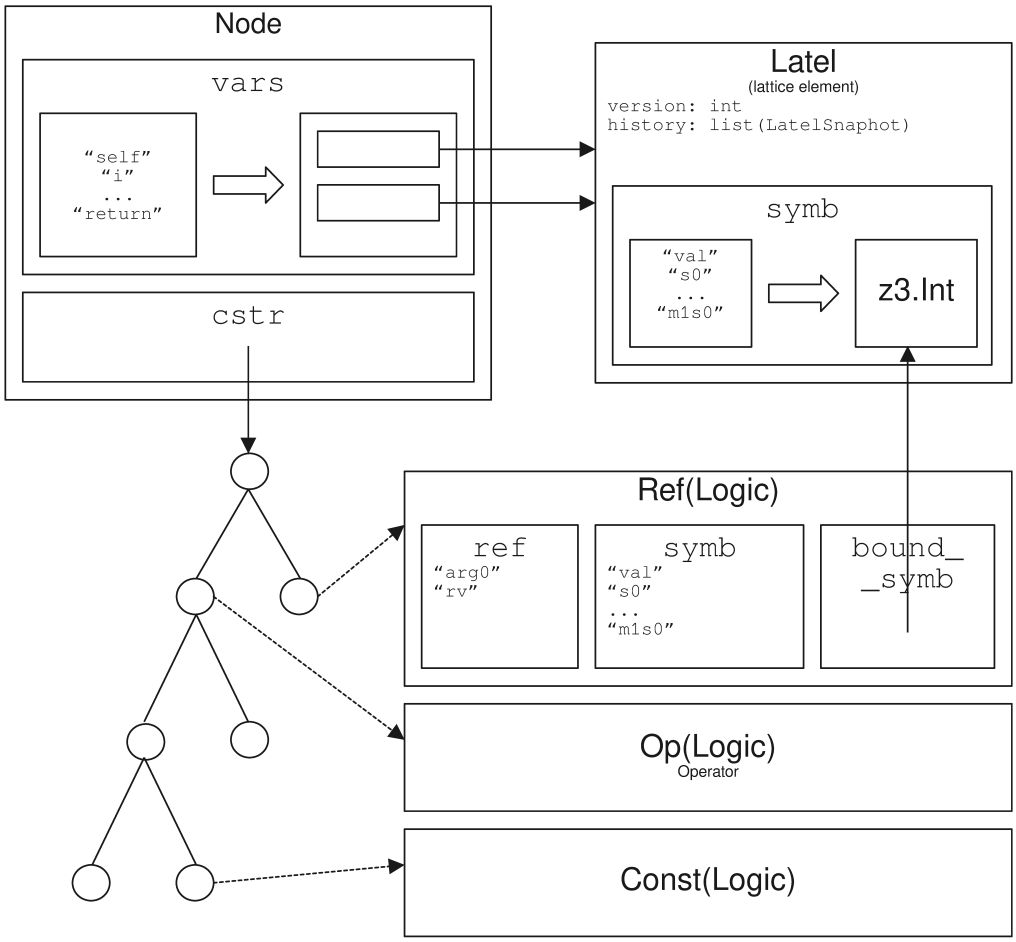

Figure 6.2 Node diagram

implemented with this approach up to some depth, they are currently not. This approach is unsustainable if the number of allowed dimensions or the number of elementary lattice elements grows. Only lengths one and two are implemented.

Slices are represented by a class. It contains a symbol that determines if a boundary is a number or not. This symbol is set to a one or a zero.

Values $\top$ and $\bot$ are represented with a class. The value $\top$ contains the union of all other symbol names. This allows to us have the information that an unknown variable has the same value throughout the program even if we don't know what type is it.

6.3.2 Symbols in lattice elements

Symbols shouldn't be changed in a lattice element from outside. The creation of new symbols in join and function recipes is handled by the lattice element. It also stores the old ones using `LatelSnapshot`.[4] When symbols are recreated (see method `repopulate`) the `version` number is incremented.

This way, the comparison of lattice elements (by the `is` operator) and the versions is enough to determine if symbols have changed.

6.4 Recipes

Recipes are represented with functions. Python treats functions as first-class objects. The recipe is created based on the node the recipe will update. The recipe function[5] binds the scope it was created in and is saved to the node.

Separate recipe functions exist for type and constraint passes.

The *function recipe* was used for the implementation of many constructs. Some other recipes wrap a call to the function recipe.

6.4.1 Ingredients

Ingredients are identified before the recipe is created. An algorithm based on DFS is used. The code for the algorithm is quite dense and is separated for readability.

Warning: The ingredients don't provide a complete list of dependencies for the relaxation of a node. This information was available in another form. Therefore, this invariant wasn't enforced. It could be helpful to enforce this invariant in the future.

[4] This code is ugly. We never expected we would need to access old symbols. If there is an opportunity in the future to rewrite this, do it.

[5] Don't confuse with a *function recipe*.

6.5 Stubs

Each function is represented with a *stub*. Each stub is a list of *stub variants*. Each variant is a tuple of the following.

- list of arguments expressed with lattice element *types* (!)
- return value expressed with a lattice element *type* (!)
- constraints represented with unparsed and unbound logic

If an argument is `⊤`, the argument will match everything. If the list of arguments is `None`, the list of arguments will match everything.

Constraints of the stub reference the lattice elements of the arguments and return values indirectly (e.g., with string "arg0", "rv").

When a stub is applied, the right variant is selected based on the actual arguments. Then, the indirect references are bound to the actual symbols referenced from the actual arguments. The resulting constraints are used.

6.6 Constraints

Constraints (abbreviated as `cstr` in code) are represented with *logic*. Logic expresses relationships between symbols and constants (e.g., if symbol A is not equal to one then symbol B must be equal to symbol C).

Symbols are global. This simplifies things as the references to symbols can just be copied from one node to another. Therefore, the constraints of a node can use symbols that aren't even referenced from the node.

Z3 is used for working with constraints. Every symbol is of the `z3.Int` type.

6.7 Logic

Manipulating Z3 expressions turned out to be unfeasible. Therefore, we created our own representation, which allows easy manipulation. We refer to it as *logic*.

Logic can exist in different forms.

- Unparsed logic is represented with a string.
- Parsed logic is represented with a tree of objects. Tree nodes are of types inherited from the `Logic` class.
- In an unbound logic, all symbols are referenced indirectly with a string that describes their role (e.g., `arg0_val`, `rv_s0`).
- In a bound logic, all symbols (`z3.Int` objects) are referenced directly.

The figure 6.3 describes how these forms are transitioned between.

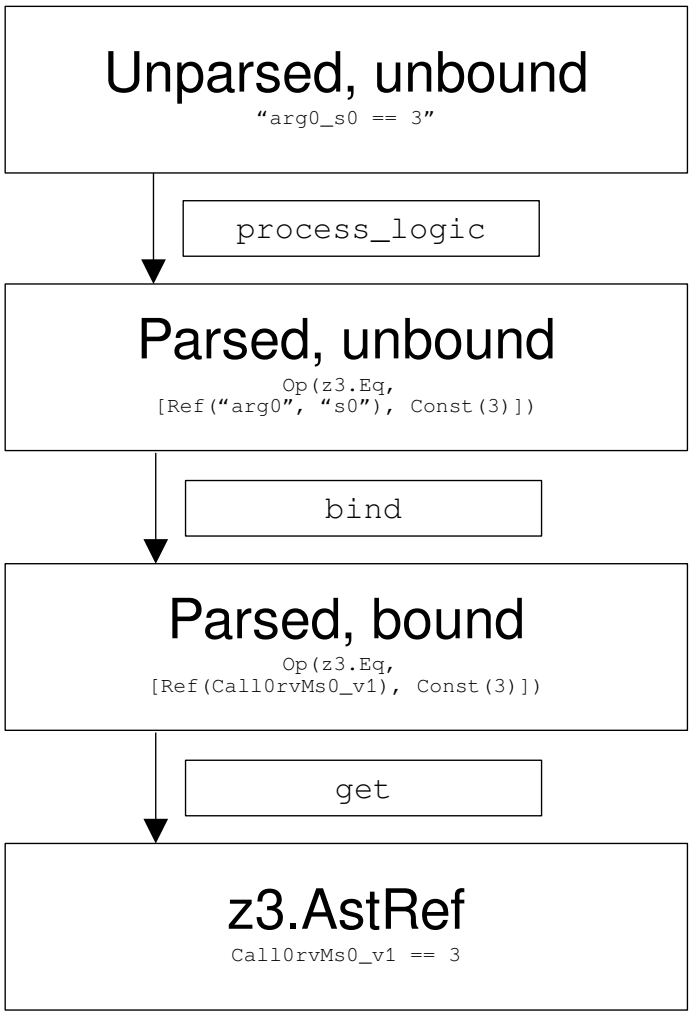

Figure 6.3 Logic diagram

6.7.1 Mainstream logic path

Unparsed logic is usually taken from the stub definition. When the stub needs to be applied, it is parsed and bound to the context of the current recipe. In this state, it goes through the program. It can still be manipulated—conjuncted or disjuncted with other logic, manually created (e.g., when processing a Constant AST node). When the constraints need to be evaluated, it is converted to `z3.AstRef` and passed to Z3.

6.7.2 Logic parsing

Unparsed logic is just modified Python. Tokenization is done fully by Python. When parsing, tokens with an underscore are replaced with a `Ref(Logic)` object. Numbers and some other constants are replaced with `Const(Logic)` objects. Some operators are replaced with others. The resulting expression is then evaluated with `eval`.

The `Logic` class redefines some operators. These operators are used to construct a tree from `Logic` objects.

Python operator precedence was cumbersome for our needs. The operators were therefore replaced in a way to introduce better operator precedence. The new operator precedence goes as follows.

- arithmetic
- `==` and `!=`
- `&`
- `|`
- `->` (z3.Implies)[6]

6.8 IDs

We use IDs to identify stubs, nodes, latels and symbols throughout the program.

- Each stub has its descriptive ID. This ID is used to search the dictionary of all stubs for the right one.
- Each AST node type has its unique name. This is determined by a big dictionary in `id_map.py`.
- Each node has its ID. It is the name of the underlying AST node type (or "Aux" otherwise) with a sequential number (counted separately for each AST node type).

[6]Yes, Python tokenizes this as an arrow operator.

- Each latel has its ID. It usually consists of the node ID it was created by, a tag that specifies where in the source code it was created (these tags don't have a system), and the ID of the lattice element type.
- Each symbol has its ID. It consists of the lattice element ID it was created by, the name of the symbols reference (e.g., `s0`[7]) and a version number.

6.9 Analysis

The job of the analysis is to determine if there is an error and find it. There are two methods to determine if a node is `sat` or `unsat`. We implemented each method as a *judge*. Each judge is a function.

One method uses heuristics to find *general matrices*. The other doesn't. By default, general matrix detection is on. To switch between the judges, the `--no_general` flag is used.

To locate an error, a *border node* needs to be found. This is just a simple iterative following of a linked list. In each node, a judge is called.

To locate the border node, a judge must be called many times. Judges are usually slow. Therefore, it is possible to turn off the location finder with the `--no_location` flag. In the future, a more effective algorithm should be used.

6.10 Python and NumPy subset

Our implementation doesn't support the full Python and NumPy specifications.

To see what Python constructs are supported, refer to `recipe.py`. It lists all the AST nodes which are supported. Other constructs are resolved with a simple recipe that just propagates all information from the `pred` ingredient. But ultimately, it is undefined behavior. To fail on undefined behavior, raise a `NotImplementedError` in the default branch of the `match` statement.

To see what NumPy functions are supported, refer to `stubs.py`. Function names are used as stub IDs. Not supported functions shouldn't result in an error. However, information about `ndarrays` may be lost as it will be resolved pessimistically.

[7]Throughout this thesis, we used `shape0` as an example of symbol reference. In our implementation, this is shortened even more to `s0`.

6.11 Debugging tools

6.11.1 Inspector

Inspector allows inspecting the state of the tree between passes and interactive browsing between states and nodes. It takes a long time to gather all the required information.

6.11.2 Visualization

Produces visualization of the recipe graph. See figure 6.4 for an example.

6.12 Module structure

- `main.py` contains a high-level algorithm.

- `ingredients.py` contains an algorithm to find ingredients.
- `recipes.py` contains an algorithm to add recipes.
- `recipe_func.py` contains the function recipe. This is used in `recipes.py`.
- `analyzer.py` contains an algorithm for the analysis of the results.

- `node_class.py` defines the monkey-patched node class.
- `lattice.py` defines the `Latel` class and lattice elements.
- `logic.py` defines the `Logic` class, and contains the logic parser.
- `stubs.py` defines all stubs.

- `inspector.py` contains the `Inspector` tool used for debugging. It is used to browse the state of the program in time.
- `vizualization.py` contains the visualization tool for debugging. It creates a visual of all nodes and ingredients.
- `id_map.py` contains the map used for creating node IDs.
- `util.py` contains helper functions.

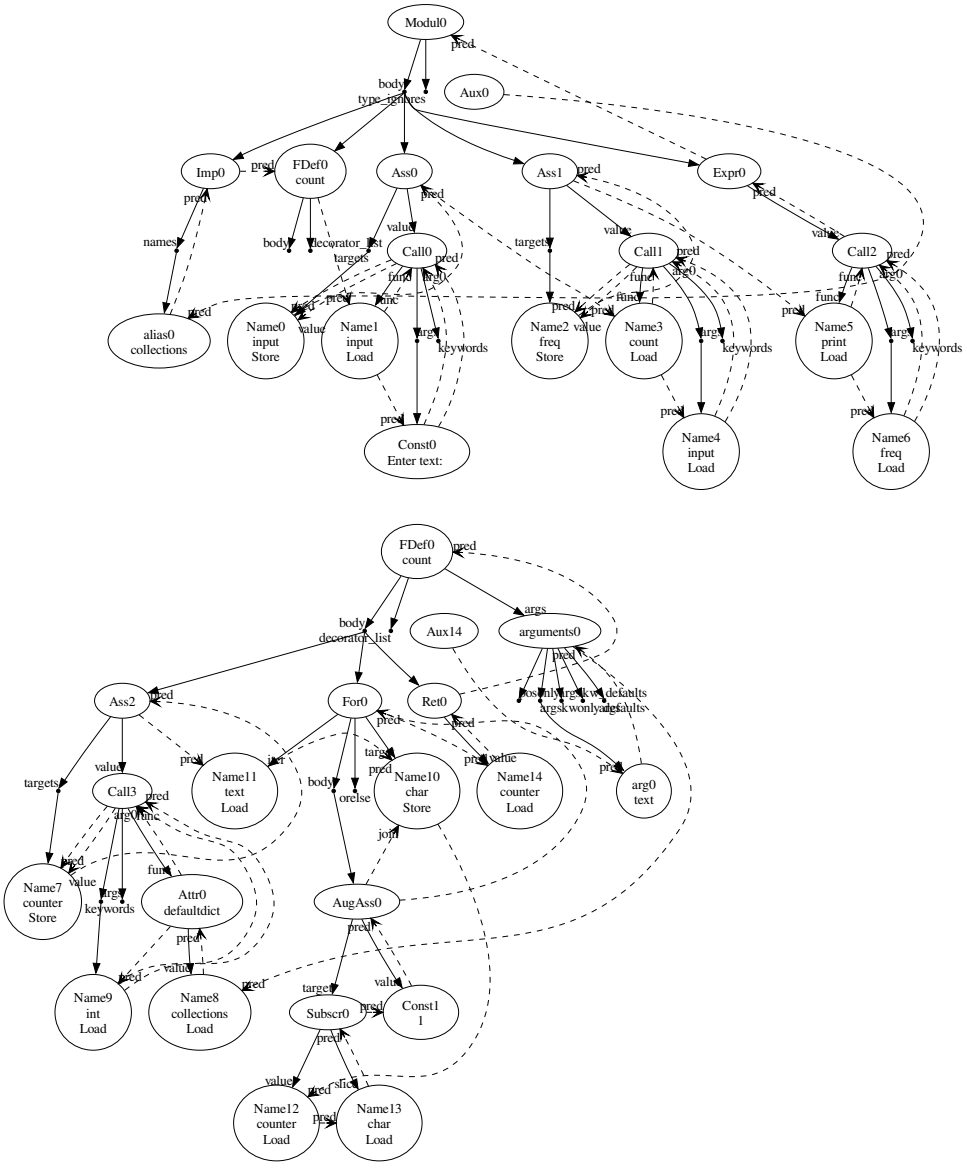

Figure 6.4 Visualization example

Chapter 7

Results

7.1 Test case `linreg`

We took code that was written for a machine learning course. It was written well before we started working on our thesis. It implements a linear regression, the well-known basic algorithm for machine learning. This code was slightly modified in the following way. The original can be found in appendix C for comparison. The resulting code can be found in listing 7.1.

- Input was loaded from a file instead of using the `sklearn` example dataset.
- The function `concatenate` was replaced with the function `hstack`. They do the same thing, but a stub for `concatenate` isn't implemented because it uses keyword arguments.
- Splitting was done manually without the help of `sklearn` functions.
- Extra code outside the main function (for testing, argument parsing, etc.) was removed.

Our program didn't report any errors in this program. Then, we started introducing artificial errors to this program. These are typical NumPy errors.

1. Use `vstack` instead of `hstack` (lines 10 and 20).

   This error was successfully detected and located.

2. Forget to add `.T` (i.e., transposition) (lines 15 and 16).

   This error was successfully detected. The program pointed to the next matrix multiplication as the point of error. Similarly, when running the program, this error would manifest itself only in the matrix multiplication.

```
import numpy as np

# In practice, these would be different
train = np.loadtxt("data.csv")
test = np.loadtxt("data.csv")
train_data, train_target = train[:,:-1], train[:,-1:]
test_data, test_target   = test[:,:-1], test[:,-1:]

# Append biases
train_data = np.hstack((train_data,
    np.ones((train_data.shape[0], 1))))

# Training
weights = (
        np.linalg.inv(train_data.T @ train_data)
        @ train_data.T
        @ train_target
        )

test_data = np.hstack((test_data,
    np.ones((test_data.shape[0], 1))))

# Testing
predictions = test_data @ weights
squares = (predictions - test_target)**2
print(sum(squares) / len(squares))
```

Listing 7.1 Test code `linreg`

3. Use `.shape[1]` instead of `.shape[0]` (lines 11 and 21).

   This error was successfully detected. The program pointed to the `hstack` function as the point of error. Similarly, when running the program, this error would manifest itself at the `hstack` function.

4. Switch the order of matrix multiplication operands.

   This error was successfully detected and located when it introduced a shape mismatch error (on line 24). In the input of the `inv` function (line 15), this wouldn't cause an error but produce an incorrect result (of the correct shape). This cannot be detected with our tool.

5. Forget a minus in a slice (twice at line 6 and twice at line 7).

   There were two cases.

   - The first minus (for construction of `*_data` variable) was missing.

     This error wasn't detected even though there is an error (matrix multiplication in the computation of predictions fails). The program is run with inputs where `target` and `test` have the same number of columns. However, our program doesn't know this information. If the number of columns in one input was different from the number of columns in the other input in just the right way, there would be no error. And since our program is conservative, no error is reported.

   - The second minus (for construction of `*_target` variable) was missing.

     This error wasn't detected because there is no error. This results in an `ndarray`, which contains more than one value. This is logically wrong, but our tool isn't supposed to detect logical errors—only shape mismatch errors.

     If the user introduced a check that the result is a single number, or the result would be used further, then it could be detected by our tool. However, our current subset of Python doesn't support `asserts` or `raise` statements.

6. Use the wrong variable name. Use `test` instead of `test_data` (line 24).

   In this case, these variables have the same shape, which we didn't expect. Therefore, this wouldn't cause an error, but produce an incorrect result (of the right shape). This cannot be detected with our tool.

   If the variable was mistaken for a variable with a different shape, an error could be possibly detected.

Overall, many errors were detected in this test case. This is our tool working at its best. There are several reasons why it performed this well.

- Only supported Python and NumPy constructs were used.
- No external libraries were used.
- Our heuristic successfully detected general matrices and utilized information about them.
- Most logic was in a single function.

This is because, in the late stages of the implementation, the subset of NumPy used in these examples was prioritized to be well-covered with the stubs. Therefore, this test case doesn't show how well our program works on average NumPy code. We are not trying to claim that it works well on average NumPy code anyway. This test case shows that our program works well on code that uses covered NumPy and Python constructs, and where our heuristics is successful.

7.2 Test case `sim`

To demonstrate the time efficiency of using our tool compared to dynamic analysis debugging, we wrote a long-running program in NumPy. It simulates a particle movement in a homogeneous field using the Euler method. See listing 7.2.

At our machine, the simulation ran for 92 seconds. Our tool finished in under a second for an error in the `plot` function and in under two seconds for an error outside a function.

When writing and testing this program, we naturally made the error of using `vstack` instead of `hstack`. Our tool detected this error. This was really nice. This error would manifest itself only after 90 seconds of runtime.

Multiple other arbitrarily introduced errors were detected, but we won't describe them in detail here. They are in the attached files.

7.3 Shortcomings

7.3.1 Python and NumPy subset

Currently, the biggest limitation of our tool is the subset of supported Python and NumPy constructs. We don't see any problems or boundaries, which would stop us from implementing the rest of these constructs. It is just time-consuming.

Some things like the representation of tuples and dimensions could be rewritten in a more general way, which could potentially save time when implementing the details.

```
import numpy as np
import matplotlib.pyplot as plt

ITERATION = 100

def plot(m):

    # Prepare the color gradient
    size = m.shape[1]
    color = np.zeros((size, 3))
    gradient = np.arange(0, size) / size
    color = np.hstack((color, np.reshape(gradient, (size, 1))))

    # Add new particle
    plt.scatter(m[0,:], m[1,:], c=color)

def step(a):
    dt = 0.0000005
    for _ in range(200000):

        # Apply acceleration on particles
        a += dt * np.array([0,-1, 0, 0, 0,-1, 0, 0, 0,-1, 0, 0])

        # Apply speed on particles
        for i in range(6):
            a[2*i] += a[2*i + 1] * dt
    return a

# Prepare
values = np.empty((12, ITERATION + 1))
values[:,0] = np.array([2, 1, 2, 2, 5, -1, 6, 2, 1, 8, 8, 3])

# Calculate
for i in range(ITERATION):
    values[:, i+1] = step(values[:, i])

# Plot
plot(np.vstack((values[0,:], values[2,:])))
plot(np.vstack((values[4,:], values[6,:])))
plot(np.vstack((values[8,:], values[10,:])))
plt.show()
```

Listing 7.2 Test code `sim`

7.3.2 Efficiency

Our implementation is quite slow. This is because we usually chose the most straightforward way of solving the problem. Some algorithms could be improved.

Namely, the border node search algorithm could use binary search to limit the number of queries to a judge.

Also, the algorithm which calls the recipes could be improved. It now generates way too many symbols which flood the recipe graph and generate a lot of unneeded constraints. By selecting more carefully which recipes should be called, this problem could be improved.

7.3.3 External libraries

Currently, only NumPy functions are modeled using stubs. However, NumPy is very often used with other libraries which extend NumPy. Covering other libraries with stubs could improve the analysis significantly.

7.3.4 Intra-procedural analysis

Currently, analysis is done in every function separately. By an inter-procedural analysis, information from outer scopes could be used to evaluate in which contexts a function is called. This could be used to determine argument types and, possibly, their number of dimensions or their shape. Similarly, the information on how a function manipulates its arguments and what it returns could be used to strengthen the analysis in outer scopes.

Possibly, this could be used to analyze external libraries.

7.3.5 Type hints

Type hints are part of Python language specification and could be used as an additional source of information. Even now, some code bases have type hints—some are just for documentation purposes; some are used by external tools like MyPy. Information that a variable or an argument is an `ndarray` could be present and used.

Moreover, more detailed information could be passed through the type hints if there was a specification describing how to express it in type hints. Then, the user could, for example, directly specify the number of dimensions or a shape.

7.3.6 General matrix heuristics

Currently, we support only one function (i.e., `loadtxt`) which returns a general matrix. Unfortunately, `ndarrays` are hardly ever created by NumPy directly.

They usually come from external libraries. This hinders better general matrix detection.

However, other means could be used to detect general matrices. Parameters of functions are good candidates, and information provided by the user could be used. This information would probably come through the means of type hints.

Chapter 8

Conclusion

We designed and implemented a static analysis tool for detecting shape mismatch errors in NumPy.

We combined data-flow analysis and symbolic execution. We track and update information about variable content, symbols that represent `ndarray` dimensions, and constraints over these symbols which arose from operations in the analyzed program. The analysis is done over a graph constructed from AST nodes. This method has proven to be very effective at collecting the required information.

Collected constraints are then evaluated with an SMT solver—Z3. Constraints are evaluated in multiple nodes to find a node where constraints become unsatisfiable. This means an error occurred at this node.

Surprisingly, very often, there exists a solution to these constraints that wouldn't occur in practice. This solution prevents us from being completely conservative at reporting errors—very few errors would be found. Therefore, we employed a heuristic to detect these unexpected solutions.

Our implementation of this method supports a subset of Python and NumPy constructs. It successfully detected errors in programs that use the supported subset. A lot of work is still needed for this implementation to reliably detect errors in typical code bases which use NumPy—the subset needs to be expanded and more effective algorithms should be used so our program runs faster on bigger inputs. However, we don't think there are any fundamental problems other than time constraints that would stop us from extending our implementation.

Furthermore, this tool could be expanded to support inter-procedural analysis and type hints. This could improve the accuracy of the analysis. Specifically on programs that use libraries that use NumPy internally.

Bibliography

[1] *NumPy User Guide*. 1.23.0. NumPy community. June 2022. URL: https://numpy.org/doc/1.23/numpy-user.pdf.

[2] *NumPy Reference*. 1.23.0. NumPy community. June 2022. URL: https://numpy.org/doc/1.23/numpy-ref.pdf.

[3] Guido van Rossum and the Python development team. *The Python Language Reference*. 3.11.1. Python Software Foundation. docs@python.org, Jan. 2023. URL: https://docs.python.org/3.11/reference/.

[4] Guido van Rossum and the Python development team. *The Python Library Reference*. 3.11.1. Python Software Foundation. docs@python.org, Jan. 2023. URL: https://docs.python.org/3.11/library/.

[5] Anders Møller and Michael I. Schwartzbach. *Static Program Analysis*. Department of Computer Science, Aarhus University, Denmark, Nov. 2023. URL: https://cs.au.dk/~amoeller/spa/spa.pdf.

[6] Erik Demaine, Jason Ku, and Justin Solomon. *Recitation 11. Introduction to Algorithms: 6.006*. Massachusetts Institute of Technology. Spring semester. 2020. URL: https://ocw.mit.edu/courses/6-006-introduction-to-algorithms-spring-2020/resources/mit6_006s20_r11/.

[7] James C. King. "Symbolic Execution and Program Testing". In: *Commun. ACM* 19.7 (July 1976), pp. 385–394. ISSN: 0001-0782. DOI: 10.1145/360248.360252. URL: https://doi.org/10.1145/360248.360252.

Appendix A

User manual

The tool was tested with Python 3.11.3. It will probably work with some older versions as well. Install package `z3-solver` from PyPI. Run with the command `python numpy-check/main.py -h`.

```
usage: numpy-check [-h] [-t MAX_TYPE_PASSES] [-m MAX_MINOR_PASSES]
                   [-M MAX_MAJOR_PASSES] [-v] [-l] [-g] [-I] [-D] [-V]
                   filename

Finds `ndarray` shape mismatch in NumPy programs. Analyzes each function
separately. Reports the location of the first error that is certain to happen.
Works only for 1D and 2D `ndarray`. Supports only subset of Python and NumPy.

positional arguments:
  filename            file to analyze

options:
  -h, --help          show this help message and exit
  -t MAX_TYPE_PASSES, --max_type_passes MAX_TYPE_PASSES
                      maximum number of type passes (if fixed point is not found)
  -m MAX_MINOR_PASSES, --max_minor_passes MAX_MINOR_PASSES
                      maximum number of minor logic passes
                      (if fixed point is not found)
  -M MAX_MAJOR_PASSES, --max_major_passes MAX_MAJOR_PASSES
                      maximum number of major logic passes (controls how much
                      information can be reached by joining before widening)
  -v, --verbose       verbosity
  -l, --no_location   don't print location of error (finding location can be slow)
  -g, --no_general    don't try to find errors for general input matrix
  -I, --inspector     debug - collect data in passes and browse internal logic
  -D, --debug         debug - enter PDB on unhandled exception
                      (used for debug prints as well)
  -V, --vizualize     debug - produce files vizualizing the graph built
                      over the AST
```

Appendix B

Structure of the attached ZIP file

```
examples
    data.csv
    linreg
        correct.py
        error-forgot_minus_1.py
        error-forgot_minus_2.py
        error-forgot_transpose_1.py
        error-forgot_transpose_2.py
        error-vstack_instead_hstack.py
        error-wrong_matmul_order.py
        error-wrong_shape_index.py
        error-wrong_variable.py
    sim
        correct.py
        error-vstack_instead_hstack.py
        error-wrong_index_order.py
        error-wrong_shape_order.py
    simple-add_biases.py
    simple-force_square-gen_matrix.py
    simple-force_square-known_shape.py
numpy-check
    analyzer.py
    id_map.py
    ingredients.py
    inspector.py
    lattice.py
    logic.py
    main.py
    node_class.py
    recipe_func.py
    recipes.py
    stubs.py
    util.py
    vizualization.py
```

Appendix C

Original `linreg` code

```
def main(args: argparse.Namespace) -> float:
    # Load the Diabetes dataset
    dataset = sklearn.datasets.load_diabetes()

    # Append a new feature to all input data, with value "1"
    data = dataset.data
    target = dataset.target
    data = np.concatenate((data,
        np.ones((data.shape[0], 1))), axis=1)

    # Split the dataset into a train set and a test set.
    data_train, data_test, target_train, target_test = \
        sklearn.model_selection.train_test_split(
                data, target,
                test_size=args.test_size,
                random_state=args.seed
                )

    # Solve the linear regression
    weights = (
            np.linalg.inv(data_train.transpose() @ data_train)
            @ data_train.transpose()
            @ target_train
            )

    # Predict target values on the test set.
    prediction = data_test @ weights

    # Compute root mean square error
    # on the test set predictions.
    squares = (prediction - target_test)**2
    rmse = math.sqrt(sum(squares) / len(squares))

    return rmse
```