GHENT UNIVERSITY

Static Analysis of Dynamic Languages

by

Harm Delva

Master's dissertation submitted in order to obtain the academic degree of Master of Science in Mathematical Informatics.

in the
Faculty of Sciences
Department of Applied Mathematics, Computer Science and Statistics

May 2017

Preface

Writing this dissertation wouldn't have been possible without some help – so some special thanks are in order. I'd like to thank the following people, in no particular order:

Prof. Dr. Peter Dawyndt for his guidance and insights while writing this dissertation.

My parents, friends, and other family for the moral support over the years.

Everyone involved at Ghent University, fellow students and professors alike, who have kept me motivated and curious.

My girlfriend – who graciously donated her submissions to be used as part of this dissertation – for everything.

Static Analysis of Dynamic Languages

Harm Delva

1 Introduction

Ghent University has developed the Dodona platform that students can use to submit solutions and receive immediate feedback. The platform makes extensive use of linter tools, which emit warnings on error prone patterns. The PyLint tool for Python has noticeable helped student avoid mistakes. This allows them to focus more on the problem solving aspect, and less on learning how to program.

This paper explores a way to give students more and better feedback. An abstract interpreter is used to reveal several common mistakes, such as type errors and uninitialized variables. The analysis is flow-sensitive and context-sensitive, as well as path-sensitive. The latter of which allows us to provide users with descriptive error messages. A data-flow analysis is performed at the same time, its results are a first step towards more advanced analysis.

2 Related Work

2.1 Static Analysis

Not every programming language helps its users to write write correct code, even though correct code is very important to developers. This is why third party tools have been developed. One category of such tools are the statis analyzers which analyze a program, either in its source code form or its compiled form, and they try to find as many candidate mistakes as possible.

Consider the code in code sample 1, which is an example from Google's Error Prone analysis tool for Java. There's a subtle mistake on line 6; even though `i` is a `short`, `i-1` is an `int`. This slips through Java's type checker because the `remove` method of the `Set` interface accepts all `Object` values – a relic from when Java didn't have generics. This subtle mistake could cause the call to `remove` to never actually remove an element. Even subtle mistakes can have serious consequences, and external tools that help catch them can be very valuable.

```
1  public class ShortSet {
2    public static void main (String[] args) {
3      Set<Short> s = new HashSet<>();
4      for (short i = 0; i < 100; i++) {
5        s.add(i);
6        s.remove(i - 1);
7      }
8      System.out.println(s.size());
9    }
10 }
```

Code sample 1: Short Set

Static languages have received a lot of attention from the static analysis community, which has lead to renowned tools such as Coverity for C and FindBugs for Java. Compared to static languages, the tool library for dynamic languages seems to be quite lacking. There are some hurdles that analysis tools for dynamic languages have to jump over to even the playing field with the static languages. Type information is an important component of many analysis tools, and this is lacking in source code written in dynamic languages. There's only a weak correlation between the attributes that an object has and its type. Individual objects can be given new attributes, separate from the type definition. This will be rare in well-written code, but analysis tools focus on badly-written code. To overcome this problem, tools for dynamic languages often use abstract interpreters [1, 2].

```
x += 1
y += 1
z += 1

x = sqrt(x)
y = sqrt(y)
z = sqrt(z)

u = x - 1
v = y - 1
w = z - 1
```

Code sample 2: Duplication

```
def foo(x):
    x += 1
    x = sqrt(x)
    return x - 1

u = foo(x)
v = foo(y)
w = foo(z)
```

Code sample 3: Refactored version of code sample 2 and 4

```
x += 1
x = sqrt(x)
u = x - 1

y += 1
y = sqrt(y)
v = y - 1

z += 1
z = sqrt(z)
w = z - 1
```

Code sample 4: Alternative order of code sample 2

2.2 Code Smells

More important than how to do the analysis is perhaps what to look for. Code smells are a taxonomy of indicators that something may not be quite right, and that some refactoring may be necessary [3]. Code smells aren't actual bugs yet, but error prone patterns. Code duplication is one of the more common ones, which immediately paints a picture of how hard it can be to automatically detect code smells.

Consider code samples 2 and 4. Despite being semantically equivalent, only the latter gets flagged as duplicated code by commercial tools such as the PyCharm IDE. In PMD, the detection tool works on a stream of tokens in the same order as they appear in the source file [4]. PyCharm's inner workings are secret, but it's plausible that it works in a similar fashion – which is why it doesn't detect the duplication.

Finding the possible reorderings of the source file would help, but this would require we know the dependencies between all the statements and expressions in the file. Luckily enough, research in the field of compiler technology has encountered similar problems, and that field of research is considerably more active. One particularly interesting approach is the *Static Single Assignment* (SSA) form [5]. In this form, every identifier only gets defined once – which makes the dependencies a lot more evident. It has proven to be a valuable form for optimizations [6], which is closely related to the refactoring code smells were designed for.

3 Fosite

There's a relatively unknown Frisian god of reconcilitation, justice, and mediation called Fosite. These are also qualities that a static analyzer should aim to have to gain a user's trust [7]. The analyzer developer as part of this paper is called Fosite, as foresight would be another good quality.

The goals were to develop an analysis tool specifically tailored to the needs of students who are new to programming. Because their submissions are short – in the order of perhaps a hundred lines – we were free to explore intensive but accurate methods. At the core lies a path-sensitive analysis that is done using an abstract interpreter. This used to provide users with a description of the cause of an error – rather than just the cause. The additional information should help with convincing users that there is an actual problem in their code, as static analyzers are often neglected due to "false positives" [7].

Apart from messages directed to the users, the analysis performs a data-flow analysis with results similar to an SSA form. The analysis runs on an *Abstract Syntax Tree* (AST), which by nature is hierarchical, so the results are hierarchical as well. This makes it impractical to store the results in the code itself – like an SSA form would do. Our results are *use-def* (or *gen-kill*) information that's being stored in a separate data structure. The results currently lack the incremental numbering that's associated with an SSA form, this is mostly an aesthetic difference. The path-convergence theorem for conversion to SSA form can easily be applied using the results from the path-sensitive analysis.
The regular SSA form is unable to handle compound structures, such as data collections, but the Memory SSA form is [8]. In this form store and load operations are handled the same way assignments and name resolution is handled in regular SSA. We combine both forms – storing dependencies to both identifiers as well as (abstract) memory locations.

4 Implementation

Since Fosite is an abstract interpreter, it uses an abstract memory, which contains abstract objects that be accessed using abstract pointers. Path objects provide the path-sensitivity by describing how certain objects can be reached. Every node in a path is defined by an ordered sequence of AST nodes that are currently being evaluated. There's a kind of path node for every notable point in a source file, such as assignments, conditionals, and function calls. Since evaluating a method may result in a variable amount of call targets, every path node also stores how many possible branches there were at every particular branch point. This is expressed more formally in definition 1. Definition 2 describes how these nodes can be ordered to form a path.

Definition 1. *A path node is of the form* $((n_1, n_2, ..., n_i), b, t)$*, where the elements* n_i *are an ordered sequence of AST node identifiers,* b *is the number of the branch that was taken, and* t *is the total of branches that were possible at that node.*

Definition 2. *Let* p *and* q *be two path nodes with forms respectively* (n_p, b_p, t_p) *and* (n_q, b_q, b_t)*,* $p \prec q \iff n_p \prec_{lex} n_q \vee (n_p = n_q \wedge b_p \prec b_q)$*.*

If we have to merge two paths at any point during execution, we'll need to make sure the paths are mergeable at all. This is important when evaluating binary operations, or during name resolution of an attribute. Definition 3 contains the definition of any given node's complementary nodes. Two paths are mergeable if neither path contains nodes that complement any other the other's path nodes.

Definition 3. *The complementary nodes of a single path node* (n_p, b_p, t_p) *are defined as* $\{ (n_p, i, t_p) \mid 0 \leq i < t \wedge i \neq b_p \}$*. If* $t_p = 1$*, an assignment node for example, there are no complementary nodes.*

Namespaces and collections are modelled in a hierarchical way, with a layer for every branch point that's currently being evaluated. This keeps changes that are made in one branch from interfering with the other branches. Once all branches have been evaluated their execution states are merged. This is when the changes made in every branch receive an extra path node – describing the conditions under which the change happened.

5 Results

To the test the accuracy and efficiency of the analysis, interesting cases have been selected from the over 800,000 submissions the Dodona platform has received over the last year. As those are a bit too long and complex to show the results of the data-flow analysis on, smaller artificial examples have been composed as well. Figure 1 contains one such example. The top left contains the code that has been evaluated, to the right of it is the data-flow analysis, and at the bottom are the results of the regular static analysis. An | has been used in the

code

```
1 y = 5
2 x = y
3
4 if 'cond':
5   y.attr = 9
6
7 x.attr
```

dependencies

```
-
y

- | y
    y

x, x.attr, 36
```

changes

```
y
x

- | y.attr, 36
    y.attr, 36

-
```

analysis

```
Error at row 7, column 1
  Object x does not have an attribute attr
  In the following cases:
  Case 1
    Assignment to y at row 1, column 1
    Assignment to x at row 2, column 1
    Condition at row 4, column 1 is false
```

Figure 1: Aliasing

data-flow results to separate the analysis of the conditional test, and the actual branches. Note the two different sorts of entries in the data-flow analysis, the 36 refers to the object on address 36. Despite being short, it shows an interesting result of using an abstract interpreter to do static analysis – a lot of aliasing information comes for free.

6 Conclusion

We have developed an abstract interpreter that can successfully recognise some common, but non-trivial, mistakes that students make. The error messages describe the circumstances in which a problem may arise so that users are more likely to agree with the analysis. Others have successfully applied similar techniques to the optimization of dynamic language [2]. Our analysis shares all the characteristics of theirs, with the addition that ours is also path-sensitive, which gives us confidence that the results can be applied to refactoring as well.

References

[1] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for javascript," in *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, (Berlin, Heidelberg), pp. 238–255, Springer-Verlag, 2009.

[2] M. Gorbovitski, Y. A. Liu, S. D. Stoller, T. Rothamel, and T. K. Tekle, "Alias analysis for optimization of dynamic languages," in *Proceedings of the 6th Symposium on Dynamic Languages*, DLS '10, (New York, NY, USA), pp. 27–42, ACM, 2010.

[3] M. Fowler, *Refactoring: improving the design of existing code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[4] T. Copeland, *PMD Applied: An Easy-to-use Guide for Developers*. An easy-to-use guide for developers, Centennial Books, 2005.

[5] B. Alpern, M. N. Wegman, and F. K. Zadeck, "Detecting Equality of Variables in Programs," in *Conference Record of the 15th Annual Symposium on Principles of Programming Languages* (J. Ferrante and P. Mager, eds.), pp. 1–11, ACM Press, 1988.

[6] M. N. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," *ACM Trans. Program. Lang. Syst.*, vol. 13, pp. 181–210, Apr. 1991.

[7] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: Using static analysis to find bugs in the real world," *Commun. ACM*, vol. 53, pp. 66–75, Feb. 2010.

[8] D. Novillo, "Memory ssa-a unified approach for sparsely representing memory operations," in *Proc of the GCC Developers' Summit*, Citeseer, 2007.

Contents

Chapter 1

Context

Ghent University has developed the Dodona platform[1] that students can use to submit solutions and receive immediate feedback. This feedback is in large part done through unit testing which lets the students know whether or not their solution is correct, but it doesn't really help them forward if it's not. To remedy this, a visual debugger is available which students can use to step through their program. Unfortunately it is limited in what it can do. More importantly, it can only let the user scroll through execution states. If an error occurred after a couple of thousand executed statements, this becomes a very tedious process.

There are ways to avoid the pain of debugging altogether. There are linter tools such as JSLint and Pylint which emit warnings on error prone patterns. The Dodona platform uses these tools as well, and relays the warnings to the students.

The Pylint tool for Python has noticeably helped students avoid mistakes. This in turn allowed them to focus more on the problem solving aspect and less on learning how to program. The goal of this dissertation is to build on top of that, giving the students even more and even better feedback. An extensive data-flow analysis can untangle even the worst spaghetti code, which makes it a prime starting point for further analysis and feedback.

[1] dodona.ugent.be – accessed 31 May 2017

Chapter 2

Related Work

2.1 Static Analysis

NASA's source code of the Apollo 11 Guidance Computer was recently digitized and put on GitHub.[1] All of it was written in an assembly language and the result worked just fine. The C programming language was developed for the Unix operating system so the latter could run on various CPU architectures [32]. In essence it's a pretty thin abstraction over assembly, in the sense that it doesn't take much pressure off of the programmers. Unix worked just fine, just like NASA's guidance controller. One could argue that programmers don't need tools to aid them – they seem to know what they're doing.

As the field of computing grew, and with it the projects, it started becoming apparent that programmers can't always rely on themselves. Even veteran developers occasionally discover a major vulnerability in their code – like the Heartbleed vulnerability in OpenSSL.[2] Of course everyone makes mistakes , but critical code should avoid them at all costs. A first line of defense against these costly mistakes is a proof of correctness. NASA's code was reliable because they had formal proofs that the most critical parts were correct [6, 14, 21]. Doing this is a massive amount of work, and a proof can still be wrong. More importantly, most verification frameworks are applied to designs and not implementations [34].

Functional programming languages are closely related to provable correctness, while also automating some of the checks. Notable examples of such languages are Haskell and ML. Both have a strong theoretical foundation and provide the programmer with a strong type system to make it easier to reason about the code. This stands in strong contrast with languages like C. While Haskell was made to facilitate writing correct code [30], C was made to be close to the

[1] https://github.com/chrislgarry/Apollo-11 – accessed 31 May 2017
[2] http://heartbleed.com/ – accessed 31 May 2017

metal and efficient [32]. The C compiler doesn't help the programmer nearly as much as the Haskell compiler. Developing correct and functional programs is obviously paramount to any programmer, so C's priorities don't always align with those of the developer.

That's where static analyzers come into play. They analyze a program, either in its source code form or its compiled form, and try to find as many candidate mistakes as possible. These mistakes are often very subtle and rare, but even a single one can ruin someone's week. Code sample 1 comes from Google's Error Prone GitHub page[3] and is a great example of how subtle serious bugs can be. The code seems to be just fine at first glance, the analysis in sample 2 reveals a subtle flaw.

```
1  public class ShortSet {
2    public static void main (String[] args) {
3      Set<Short> s = new HashSet<>();
4      for (short i = 0; i < 100; i++) {
5        s.add(i);
6        s.remove(i - 1);
7      }
8      System.out.println(s.size());
9    }
10 }
```

Code sample 1: Short Set

```
1  error: [CollectionIncompatibleType] Argument 'i - 1' should not be
2  passed to this method; its type int is not compatible with its
3  collection's type argument Short
4        s.remove(i - 1);
5                ^
```

Code sample 2: Analysis of code sample 1

Subtracting an `int` from a `short` results in an `int`. In this case it won't cause an actual bug yet, because both types use the same hashing function. But this isn't something a developer should rely on. The JVM originally didn't support generics, and their implementation is still a bit rough. The `remove` method of a `List` instance accepts any `Object` instance. This can result in calls that never actually remove anything. If that call happens to only occur in a corner case in the 1000th iteration of a loop, this can lead to some very confusing bugs.

[3] https://github.com/google/error-prone – – accessed 31 May 2017

2.1.1 Popular Languages

Every practical programming language is Turing complete, so in theory they should all be equal. This is a misconception that has been called the Turing tar-pit [44]. Everything is possible in a Turing tar-pit, but nothing of interest is easy. Programming languages where things of interest are perceived to be easy can be considered powerful. These languages are often the ones with lenient compilers or runtime environments such as C, Python, Javascript, … In other words, languages that don't get in the way of the programmer too often. These are also by far the most popular languages.

As illustrated in the previous section, this may not always be a good idea as humans tend to glance over subtle details. This makes the need for additional tools very important for any project that aims to achieve high reliability. The alternative is long and painful debugging sessions. At some point these languages no longer make it easy to do things of interest.

C

The C programming language has been one of the most popular programming languages for a couple of decades now. Depending on who you ask, the best thing about C is either its efficiency or its simplicity. The latter is what gives developers the power they desire. This comes at a cost however; with great power comes great responsibility.

Let's focus on the other main attraction of C, its efficiency. This comes at a cost as well and it's one many people forget about. C's *raison d'être* isn't making developers feel good about themselves, it's generating efficient code. It was created to be a thin abstraction layer over various assembly languages that were limiting software development at the time [32]. This has left some holes in the C specification; for example, not all CPU architectures handle a division by zero, so the C specification doesn't specify what to do in this case.

Undefined Behavior

There are a lot of things the C specification doesn't specify a behavior for, which leads to undefined behavior. Some are well-known, such as using freed memory. Others catch people by surprise. For example, the GNU libc website claims that `1/0` yields `inf`,[4] even though the C99 specification clearly contradicts them [15]. The C99 standard introduced the `INF` macro, but it doesn't specify which operations should result in one. Division by zero is still as undefined as it has always been.

[4] http://www.gnu.org/software/libc/manual/html_node/Infinity-and-NaN.html – accessed 31 May 2017

Entire papers have been written on the subject of undefined behavior [35, 40, 49]. One striking thing is how recent a lot of these papers are. Even though the language is over 40 years old, this is still an active field of research. Compilers are getting more advanced and with it the optimizations they perform. Some of those optimizations rely on the fact that the compiler is under no particular obligation when optimizing code containing undefined behavior.

```
1 if (arg2 == 0)
2   ereport(ERROR,
3     errcode(ERRCODE_DIVISION_BY_ZERO),
4     errmsg("division by zero"));
5
6 /* No overflow is possible */
7 PG_RETURN_INT32((int32) arg1 / arg2);
```

Code sample 3: Undefined Behavior

Code sample 3 was part of PostgreSQL [35]. The call to `ereport` on line 2 never returns. It does some logging before calling `exit`. In the mind of the developer this prevents the division by zero on line 7. Looking at the code on GitHub,[5] the function this sample came from indicates that calling it will return a `Datum` struct. According to the language specification, this function *must* return a value each time it's called. The body of the null check does not return anything, so the compiler concludes that the rest of the function will also be executed and division by `arg2` will always occur.

Division by zero is also undefined behavior in C, so the compiler concludes that `arg2` won't ever be zero – it wouldn't get used in a division otherwise. As a result, the null check gets flagged as dead code, and is removed entirely.

```
1  if (arg2 == 0)
2  {
3    ereport(ERROR,
4        (errcode(ERRCODE_DIVISION_BY_ZERO),
5         errmsg("division by zero")));
6
7    /* ensure compiler realizes we mustn't reach the division
8    (gcc bug) */
9    PG_RETURN_NULL();
10 }
```

Code sample 4: Fixed Undefined Behavior

[5] https://github.com/postgres/postgres/blob/master/src/backend/utils/adt/int.c#L847 – accessed 31 May 2017

Code sample 4 contains the fixed code. By adding an explicit return (in the form a macro), the compiler leaves the null check intact. Notice how the comment blames a "gcc bug". This illustrates how even experienced developers seem to misunderstand their language of choice.

Tools

Not a single other programming language comes close to having as many external tools as C (and by extension C++). Many developers heavily depend on these tools in their usual workflow. One of the most established toolsets are the ones in Valgrind.

Valgrind is a suite of dynamic analysis tools, the most famous of which is Memcheck. Memory related bugs are some of the hardest to track down because they're ultimately undefined behavior. For example, using a memory location after freeing might not always crash the program. There's an informal term for bugs like these: *heisenbugs*. Something might go wrong but when you try to isolate the cause everything seems to be just fine. Especially since Address Space Layout Randomization (ASLR) tends to be disabled during debugging but not during normal execution.

This is where Memcheck comes into play. It analyses the program during regular execution and keeps track of what happens to the memory. This way it can notice memory related bugs such as use-after-free and report them back to the developer. Unfortunately it's not a perfect solution. There can be false positives as well as false negatives, and it is quite incompatible with some libraries such as OpenMPI [45].

A lot of companies rely on analysis tools to manage their large C projects, and when there's demand in a market, the supply will follow. There's an impressive amount of commercial analysis tools available. Coverity is one of the most established ones.

Dawson Engler, one of Coverity's co-founders, is one of the leading researchers in the field of static analysis. He also co-authored a great paper in which he describes how difficult static analysis is in the real world [4]. One particularly interesting part of the paper explains that there's a fundamental misunderstanding of what programming languages and compilers are. A programming language can exist as an abstract idea or a piece of paper. While the language a program is written in is whatever the compiler accepts. In other words, compilation is not certification. A good first step for a static analysis tool is to make sure that the input adheres to the specification. They go on to pose the hypothetical question: "Guess how receptive they [the end-users] are to fixing code the "official" compiler accepted but the tool rejected with a parse error?"

Even with a plethora of tools available, C remains an untamed beast. Some tools like Valgrind are great at what they do but are still limited. Other tools like Coverity seem to fight stubborn developers as often as they fight bugs.

Dynamic Languages

According to Guido Van Rossum, he designed Python because he wanted to make a descendant of ABC that would appeal to Unix/C hackers [23]. This is a bit worrying considering the previous section, as C is a remarkably hard language to analyze. Javascript's situation isn't great either. There are no namespaces, no modularization system, no classes, no interfaces, no encapsulation, and so on. Eric Lippert, who was on the ECMA committee during the early days of JavaScript, has a Stackoverflow post where he discusses why Javascript is so ill-fit for static analysis and programming in the large [43]. Even though Stackoverflow is a dubious source for any academic work, the author's experience should make up for it.

Both Python and JavaScript are popular languages, so one might expect there to be a good amount of analysis tools for these languages as well. There seem to be two classes of analysis tools available for these languages right now: linters and type checkers. The linters are very popular, but as a later section will discuss they focus on error prone patterns instead of errors, and they prioritize efficiency over in-depth analysis. Type checkers are less popular, mostly due to circumstances.

A type analysis tool for Javascript called JSure was developed in 2009, in part because the authors felt a need for deeper analysis than what JSLint could offer [16]. They utilize an abstract interpreter to perform a flow- and context-sensitive analysis. Typescript was released a few years later, which fills a similar function as JSure, but with the researching power of Microsoft behind it. Python has a similar tool called MyPy which takes a fundamentally different approach [17]. Python 3.5 introduced type hints for analysis tools such as MyPy to use. Although promising, the tool is plagued with a few limitations. For starters, it only supports Python versions since 3.5, while the users who stand the most to gain from it are the ones with large code bases – which are predominantly on version 2.x for legacy reasons. On top of that, most existing Python 3 code does not even use type hints at this point, so stand nothing to gain from MyPy.

What these two classes of analysis tools for dynamic languages have in common is that they're not as in-depth as the ones for static languages. Tools like Coverity for C and Error Prone or FindBugs for Java go as far as detecting racing conditions, while the tools for Javascript and Python are still experimenting with type analysis. This isn't because dynamic languages aren't important, they're used to power some of the biggest websites [13, 18], it's just a hard thing to do. The following anecdotal examples illustrate why.

Classes

Lots of developers are familiar with class hierarchies and like using them, but there is one glaring difference between how static languages and dynamic languages handle classes. In most static languages, the class definition declares which methods and attributes are defined for objects of that class. This isn't the case with dynamic languages. Adding a method to an object is as simple as adding a callable object to the class's namespace. This is bad news for static analyzers that try to do type inferencing on dynamic languages. Consider the following bug that occurred while I was working on the Dodona platform.

The Ruby builtin `String` class provides a `split` method that does the same thing as in most other languages: its argument is some separator that is used to cut the `String` object into an array of smaller `String` objects. While writing a function that takes a few strings as arguments, the strings had to be split on whitespace. Calling the `split` method on one of the strings returns the same object instead. Using `puts` on that object just shows the same exact `String` we wanted to split, with all the whitespace still there, weird. The documentation says that it should work. Testing the method in a REPL (Read–eval–print loop) environment confirms that the `split` method should indeed split the `String`. A few confused hours later, it turned out the object wasn't a `String` but an `Array` of `String` objects. The Ruby documentation doesn't mention that `Array` defines a `split` method. Googling "ruby array split" refers to the Ruby on Rails documentation. A library silently declared a method on a builtin type, and `puts` prints every element of an `Array` on a separate line, so it just prints a single `String`.

This approach to classes has some serious implications. Not only does it confuse newcomers, it confuses static analyzers as well. As these languages typically don't have type declarations, a logical alternative would be type inference. But consider the following case, `split` is being called on something that's either an `Array` or a `String` at the current stage of inferencing. A regular Ruby analysis tool might conclude that the object must be a `String`. Analyzing Ruby on Rails code would either require the tool to analyze the entire framework's code first to learn that it adds a `split` method to `Array`, or it might even require a custom analyzer for Rails altogether.

This is why static analysis of dynamic languages usually starts with an abstract interpreter [12, 16, 24]. You don't have to infer the type of an object if you know where it came from.

Refactoring

Python's lack of structure makes it hard to work incrementally. At some point during implementation of the interpreter developed for this dissertation, line and column information had to be added to the AST structure. Python's `ast` module contains everything one could need

to work on Python's AST, including `lineno` and `col_offset` for seemingly all classes. With the exception of a few classes, such as `generator`. Implementing generators didn't come until much later, long after the analyzer started relying on the existence of `lineno` and `col_offset` for every node.

Refactoring dynamic languages is a challenge. What if we want to change the name of a method called `add` to `insert`. Can we replace all occurrences of `.add` to `.insert`? That might change other classes as well. As discussed in the previous section, type inferencing is non-trivial. Even IDEs that are renowned for their ability to refactor code, such as the JetBrains IDEs, rely on manual feedback to filter out the false positives. Reporting false positives is not always an option however. That's what causes people to question your tool [4].

`NaN`

There are problems that plague both static and dynamic languages, such as the existence of `NaN`. Some languages like Python and Java do a good job at preventing `NaN` from entering your program through raising exceptions, but once they do find their way in, you're left at the mercy of the IEEE 754 standard. The most recent version of the standard is IEEE 754-2008 and was heavily influenced by the ISO C99 standard, which introduced `NaN` and `INF` to the C language.

Since this standard, if one of the function's arguments is `NaN` but the other arguments already determine the function's value, the result is no longer `NaN`. For example, this means that `pow(1, NaN)` is equal to `1`. Mathematically this is at least a bit dubious, $1^{0/0}$ shouldn't be defined if $0/0$ isn't either. The C99 standard introduced various other oddities [15]: `pow(-1,` $\pm\infty$`)` is `1` because all large positive floating-point values are even integers, and having a `NaN` in the imaginary part of a complex number does not matter when the value is being cast to a real number – it's apparently possible for a value to be only slightly undefined.

It has become normal for `NaN` values to somehow disappear as if nothing was wrong, leading to some very confusing bugs. A fellow student ran into the consequences while implementing his own dissertation (personal correspondence). The Theano framework[6] has a function to compute the gradient of some expression. A `NaN` found its way into the input of this function but the result somehow didn't contain a single `NaN`. It became seemingly random noise instead. When implementing something computational, any person's first instinct would be that the algorithm is wrong.

Static analysis should be able to help alleviate this problem. If the floating-point value that's being used came from a `sqrt` function, there should be an `isNaN` check first. Alternatively, the Rust

[6] http://deeplearning.net/software/theano/tutorial/gradients.html – accessed 31 May 2017

programming language tries not to hide the fact that floating-pointing values can be `NaN`. Without providing your own implementation of the `Ord` trait for the `f64` type, it's impossible to order a vector of `f64` values because comparing two `f64` values might be meaningless. It does however provide an implementation of the `PartialOrd` trait, which returns an `Option<Ordering>`, which makes it explicitly clear that the result can be `None`.

2.1.2 Safe Languages

Some languages try to protect their users against themselves. Most of these languages are functional languages but there are notable exceptions such as Ada.

Haskell

Having strong ties to the lambda calculus [30], Haskell is the archetype of a safe language. Like in most functional languages, all state in a Haskell program is implicit and by extension there are no side-effects. One of the core concepts of the language is that Haskell code should be easy to reason about. That's why this language deserves a section in a dissertation about static analysis and data-flow analysis; Haskell's design makes these things pleasantly simple.

```
1  def reset(arg):
2    arg.attr = None
3
4  x = A()
5  x.attr = 4
6  y = x
7
8  reset(y)
9
10 x.attr += 1
```

Code sample 5: Aliasing

Consider the Python code in Code sample 5. Knowing that `x` and `y` refer to the same object is integral to realizing that `x.attr += 1` will result in a type error, as `x.attr` is `None` since the call to `reset`. Haskell has no side-effects – function calls such as `reset(y)` on line 8 wouldn't be able to change anything. Additionally, it has no explicit state and thus no assignments and no aliasing to begin with. This trait can be mimicked in other languages using the Static Single Assignment (SSA) form where every variable is only defined once. Section 2.3.1 will discuss this form and how it relates to code analysis.

Rust

A newer language that favors safety over accessibility is Rust. While it does have explicit state, Rust also makes the aliasing problem trivially easy for static analyzers. With some exceptions like the reference counting types, every piece of memory has a single owner in Rust – which means that by default there's a single way to access any piece of data. This prevents aliasing problems like the one in sample 5 because after executing `y = x`, `x` is no longer a valid identifier – its data has been moved to `y`. On top of that, Rust has explicit mutability. This concept came from C/C++ where it's good practice to label all immutable things with `const`, except that Rust does it the other way around. This means that unless `y` was declared to be mutable, the assignment to `y.attr` wouldn't compile either.

In languages like Java that heavily advocate encapsulation, it's not uncommon to write something like a `List<Element> getElements()` method, so that other modules can see which data is present. Returning the original list would mean anybody can change its contents. That's why it's considered good practice to return a deep copy of the list instead. Deep copies carry a significant overhead with them, so developers end up choosing between a safe architecture or an efficient implementation. Rust lets data be borrowed, so that other modules can see their contents. The result is a borrowed reference which is very similar to a pointer in C, with the exception that borrowed references carry a lot more type information with them. For starters, it's possible to borrow something mutably or immutably. If there is a single mutable borrow to an object, there can't be any other borrows at the same time. This is a mechanism that's supposed to avoid unexpected side-effects. Another thing that's part of a borrow's type is its lifetime. If object A wants to borrow something from object B, B has to exist at least as long as A. This mechanism directly prevents all sorts of memory related bugs that occur in C.

Rust is an interesting example of the relationship between languages (more specifically the compilers) and static analyzers. The Rust compiler enforces a lot of things that the C compiler doesn't, but that the C community has written their own tools for. One might wonder why C is even still around if Rust seems to be a more complete package. Part of the answer is that Rust is a very restrictive language, leading to frustrated developers. Keeping in mind Coverity's paper [4], it's a lot easier to ignore an analyzer than it is to fix a compile error.

In fact, this can be seen as an instance of the loosely defined concept of an *XY problem*.[7] If a person knows how to do something using method X, he's less likely to learn method Y – even if method Y is clearly the better choice. Rust has received a lot of criticism from renowned developers for ridiculous reasons, such as being unable to have two mutable references to the same object. That's how they would do it in C, so it must be the right way, even though `Alias`

[7] https://manishearth.github.io/blog/2017/04/05/youre-doing-it-wrong/ – accessed 31 May 2017

`XOR Mutability` is a common design principle. This problem is relevant in this dissertation for two reasons. Static analysis tools run into the same mentality issues [4], and it shows that it's important to never pick up bad habits in the first place. The field of application of this dissertation is ultimately helping students learn how to program, before they have a chance to grow any bad habits.

2.1.3 Tools

Linters are a class of static analyzers that are particularly interesting in the context of this dissertation. They're made to report error prone patterns during development to prevent actual errors, which is what sets them apart from other analysis tools. In order to do this they have to be very efficient, so that feedback can be given while the code is being written. Other analysis tools such as Coverity are usually only run nightly, or once per commit alongside unit testing [4]. This also means that the analysis they're able to do is usually quite shallow, and purely syntactic. That doesn't make them any less useless however, they just report on generally bad patterns such as shadowing builtin function or writing long functions. One of the most renowned linting tools is JSLint (and its successors JSHint, ESLint), its Github page contains the following wisdom which describes the underlying philosophy quite well [46]:

> "The place to express yourself in programming is in the quality of your ideas and the efficiency of their execution. The role of style in programming is the same as in literature: It makes for better reading. A great writer doesn't express herself by putting the spaces before her commas instead of after, or by putting extra spaces inside her parentheses. A great writer will slavishly conform to some rules of style, and that in no way constrains her power to express herself creatively."

Few developers should find anything wrong in that reasoning. The main point of contention is which rules to follow. Crockford is a proponent of using certain language features sparsely [9]. In his book titled Javascript: The Good Parts he describes which features of Javascript he deems good (or great and even beautiful) and which parts should be avoided. The bad parts are everything that lead to error prone code in his experience. The usual example is the `switch` statement. He has given a presentation at Yahoo where he gives more examples of why JSLint discourages the patterns that it does [42].

All analysis tools serve a common purpose, but are ultimately still very diverse. Static and dynamic analysis tools do things very differently and even within static analysis tools there's a lot of variation. Linters work mostly syntactical, focusing on speed and immediate feedback.

Other tools like Coverity do a much deeper analysis and usually run nightly [4]. Another major difference among static analyzers is soundness. Sound analysis tools are based on formal logic and more generally more comprehensive, but slower and more prone to false positives [38]. Analysis of dynamic languages relies heavily on abstract interpreters [12, 16, 24], and the closely related domain of symbolic execution which is discussed in section 2.4 is applied to all languages for automated unit testing.

With all these different approaches to the same problem, one might wonder which is the best. Unsound methods seem to come out on top as most practical languages are unsound themselves [22]. Others simply say that it doesn't matter how you do it, as long as the results are good [4]. Every approach has its own pros and cons, and users can always use multiple ones to get the best possible coverage.

2.2 Code Smells

More important than how to do the source code analysis, is perhaps what to look for. Code smells are a taxonomy of indicators that something may not be quite right in the code, and the term was coined in a book on refactoring [11]. In that book code smells are indicators that some refactoring might be necessary. They're aimed at software architects, and are informal descriptions of things that may induce technical debt. They're not actual bugs yet but error prone patterns, the sort of things linters aim to detect. One of the code smells is *cyclomatic complexity*, i.e. having too many branches or loops. This is something linters also detect, but it lacks the refactoring aspect of the code smell.

Code smells were meant to be indicators, so that professional software architects knew where to focus their refactoring efforts. This may not be sufficient when trying to help new programmers, as they might need help knowing what and how to refactor. There are also code smells that linters currently do not pick up because they would require deep analysis. The most notable of which would be the *Code Duplication* smell. JetBrains has developed some of the best refactoring IDEs such as IntelliJ and PyCharm, but started off by developing code refactoring tools such as IntelliJ Renamer for Java and Resharper for C#. These are made for large commercial code bases, and are ultimately advanced heuristics that still miss obvious duplication. PyCharm 2016 was unable to find the duplication in code sample 6, though it should probably be refactored to something like code sample 7.

```
x += 1
y += 1
z += 1

x = sqrt(x)
y = sqrt(y)
z = sqrt(z)

u = x - 1
v = y - 1
w = z - 1
```

CODE SAMPLE 6: Duplication

```
def foo(x):
    x += 1
    x = sqrt(x)
    return x - 1

u = foo(x)
v = foo(y)
w = foo(z)
```

CODE SAMPLE 7: Refactored version of code sample 6

```
x += 1
x = sqrt(x)
u = x - 1

y += 1
y = sqrt(y)
v = y - 1

z += 1
z = sqrt(z)
w = z - 1
```

CODE SAMPLE 8: Alternative order of code sample 6

JetBrain's tools are closed source so it's unclear whether or not code sample 6 was deemed too simple to refactor. Assuming they work in a similar fashion as competing tools however, it's the order of statements that causes it to fail. Tools like PMD, Duploc, and CCFinder all work on a token stream in the same order as it appears in the file [8, 19, 25]. Code sample 8 illustrates how sample 6 could be reordered so that tools can detect the duplication just fine. Compiler technology is a much more active field of research than duplication detection, and the problem of reordering instructions occurs there as well. One of their solutions is discussed in section 2.3.1, which discusses the Static Single Assignment (SSA) form.

When analyzing student code, this can be a serious limitation. Consider code sample 9, which just reads 4 numbers from `stdin`, increments them, and stores them in `x`. Even if this duplication gets detected, most tools are targeted at professional developers who would never write code like this. The critical difference is that the refactoring shouldn't introduce a new function but a loop such as in code sample 10.

There are some other code smells besides code duplication that could be interesting for new programmers. The following list contains some that at first glance look like the most promising.

- *Large class*: A class that's grown too large and probably has too many responsibilities.
- *Long method*: Much like the previous one, this method probably has too many responsibilities.
- *Inappropriate intimacy*: When a class heavily depends on implementation details of another class.
- *Cyclomatic complexity*: Too many branches, also informally referred to as *spaghetti code*. Linters do a fair job at pointing them out but offer little help in fixing them.

```
x = []

s = int(input()) + 1
x.append(s)
s = int(input()) + 1
x.append(s)
s = int(input()) + 1
x.append(s)
s = int(input()) + 1
x.append(s)
```

CODE SAMPLE 9: Duplication

```
x = []

for _ in range(4):
  s = int(input()) + 1
  x.append(s)
```

CODE SAMPLE 10: Refactored version of code sample 9

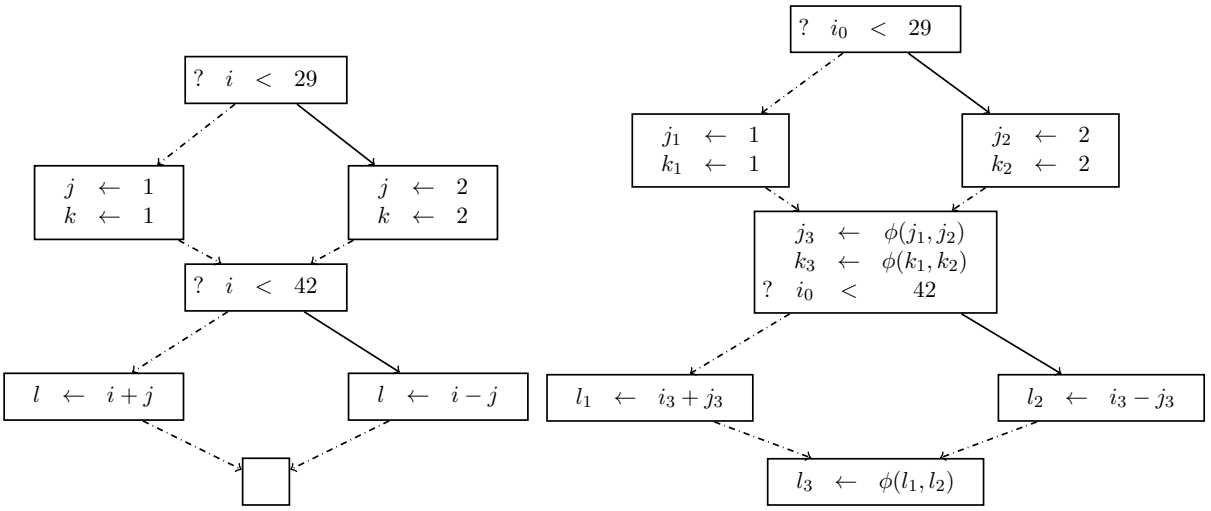
2.3 Compilers

Static analyzers aren't the only tools that aim to make code better – compilers do so as well. Refactoring from a software architect's point of view is aimed at making the code easier to read and maintain. Optimizations are aimed at making code more efficient. Optimization and refactoring sometimes do opposite transformations, for example in sample 10 where unfolding the loop results in the code in sample 9. Both operations transform code in a conservative manner though, i.e. without changing the semantics (of valid code). Optimization is a very active area of research, with companies like Google and Apple working on the LLVM,[8] Oracle on the JVM, Red Hat on the GCC,[9] … More importantly, even the most esoteric features in compilers have proven their value as they're part of a real product, which gives confidence that an analyzer that uses the same principles will work as well.

2.3.1 SSA

Static Single Assignment (SSA) form is a program representation that's well suited to a large number of compiler optimizations such as conditional constant propagation [36], global value numbering [2], and code equivalence detection [37]. It was introduced in a paper by IBM [2] in the 80s but had little practical success initially. It wasn't until several optimizations were found [7, 10] that it started becoming popular. Since then it has found its way into most popular compilers. LLVM has used it in its virtual instruction set since its inception in 2002 [49], it's what powered Java 6's JIT [20], and it's what's behind recent GCC optimizations [27, 28].

[8] http://llvm.org/foundation/sponsors.html – accessed 31 May 2017
[9] https://gcc.gnu.org/steering.html – accessed 31 May 2017



(A) Original form (B) SSA form

FIGURE 2.1: SSA transformation

The idea is very simple, every variable can only be assigned to once. This can be done by adding incremental numbers to each variable. For example, `x = x + 1` becomes $\texttt{x}_1$ `=` $\texttt{x}_0$ `+ 1`. This is a trivial transformation for straight-line code, but becomes a bit harder when dealing with branches. Figure 2.1 illustrates how this is solved. Every branch introduces new variables, and a *ϕ-node* gets inserted where paths converge. This node mimics a function call that can return either of its two arguments. This doesn't correspond to an actual function call, it's just a token that gets inserted to help with the analysis.

Memory SSA

As cited in the beginning of this section, the SSA form has a lot of benefits, but is limited to scalar values and is not well suited for compound structures such as arrays. One of the solutions to this problem is Memory SSA as implemented by GCC [27]. This is a transformation that runs on a sufficiently low-level representation of the source code, where there's a notion of `LOAD` and `STORE` instructions.

Because the actual memory locations are not known during compilation some abstractions are needed. GCC uses compiler symbols which they called tags to represent regions of memory along with two virtual operators `VDEF` and `VUSE`. For every `LOAD` a `VUSE` of some tag(s) gets inserted, and likewise for `STORE` and `VDEF`. There are three types of tags:

- *Symbol Memory Tag* (SMT): These are the result of a flow-insensitive alias analysis and are almost purely type-based. For example, all dereferences of an `int*` receive the same tag to reflect the fact that they might point to the same memory location.

- *Name Memory Tag* (NMT): These are the result of a points-to analysis applied after the program is in a SSA form and inherits their flow-sensitive properties [27]. In other words, the GCC uses multiple different SSA forms.
- *Structure Field Tags* (SFT): These are symbolic names for the elements of structures and arrays.

Tags get used in a similar way as variables in the regular SSA algorithm. Assigning to a tag is like assigning to all symbols that have received that tag. In the original implementation, *call clobbering*, i.e. a function call that alters the state of one of its arguments such as in code sample 5, is handled the same way as global variables [27]. All arguments that are known to get clobbered sometimes are put into a single set. Calling functions that would clobber one of them is assumed to clobber all of them. This pessimistic approach has been replaced in recent implementations, where all related code has been rewritten to use an *aliasing-oracle* [41].

LLVM's usage of Memory SSA is not quite as documented. Their documentation refers to GGC's paper [27, 47] and mentions that "Like GCC's, LLVM's MemorySSA is intraprocedural.". As mentioned in the previous section, this isn't entirely true for GCC anymore. It doesn't seem to be true for LLVM either. A recent publication describes *Static Value-flow* analysis which produces an interprocedural SSA form [33]. It has been part of LLVM since 2016 and like GCC's implementation it uses the results of an external points-to analysis.

2.4 Symbolic Execution

Rather than using concrete values, symbolic execution uses symbolic values. Those values represent possible values through a set of constraints. This technique is commonly used to find interesting corner cases for unit testing [5, 29, 31] and the original SSA paper used similar principles to make their analysis more precise [2].

At each branch point a *path constraint* gets introduced, which is a symbolic expression that must be true for the current branch to be taken. If there are no values that satisfy all the constraints, the branch gets flagged as dead code. In the case of a static analysis tool this will most likely result in a warning, while a compiler would just remove the dead branch. For example, consider code sample 11. The condition `y > 0` becomes a constraint during the execution of the positive branch, as well as `x > -1`. The former constraint is pretty simple to add, the latter requires some serious bookkeeping. Which implies a close relation between symbolic execution and data-flow analysis.

Symbolic execution is a very powerful tool but comes with a few limitations. One is the *state explosion problem*. The number of paths can increase very fast. The Java Pathfinder manages

```
x = int(input())
y = x + 1

if y > 0:
    z = sqrt(y)
else:
    z = 0
```

Code sample 11: Symbolic Execution

this problem using state matching and backtracking [29]. State matching will check if a similar state has already been processed. If it has, it will backtrack to the next state that still has unexplored choices.

An advanced symbolic executor for Python is PyExZ3 [1, 3], which is built on top Microsoft Research's Z3 theorem prover. Z3 is a remarkably good fit to reason about programming languages and even supports strings and other sequences [26]. There are even extensions that add support for regular expressions and other advanced constraints [39]. As another team at Microsoft Research has pointed out however, sound methods such as symbolic execution tend to run into problems when applied to unsound languages [22]. Z3 heavily relies on type information, which is not present in Python code. Lists and dictionaries are still open challenges [3], as Z3 requires an element type upon declaring a sequence [26].

Symbolic executors typically focus on scalar values. Collections tend to be a lot harder to analyze. Consider code sample 12 for example, which defines a function `foo` that prints `homogeneous` if and only if its argument is a collection and all elements in that collection are of the same type. The positive branch introduces a constraint on the length of `x`. Z3 can already handle this quite well. The relation between uniqueness and the length of a set is considerably harder though, and it's a very *pythonic* pattern.

```
def foo(x):
  if len(set(map(type, x))) == 1:
    print('homogeneous')
  else:
    print('heterogeneous')
```

Code sample 12: Symbolic Execution Challenge

Chapter 3

Fosite

There's a relatively unknown Frisian god of reconciliation, justice, and mediation called Foseti. These are also some qualities that a good static analyzer should aim to have as well to gain a user's trust [4]. The analyzer developed as part of this thesis is called Fosite,[1] as foresight would be another good quality.

3.1 Goals

Unlike most existing tools, Fosite's focus is on analyzing small pieces of code submitted by students that are new to programming, and this comes with both advantages and disadvantages. The biggest advantage is that we're free to explore slow but accurate methods. Not entirely free however as feedback should still be fast. Imagine being a stressed out student, working on your final exam and every submission takes a minute to run because submissions come in faster than they get processed.

The most important requirement of all is that all warnings have to be as helpful as possible. Providing the right details should make the detection of errors more convincing and will be met with less resistance from the user [4]. This is doubly important for new programmers; simply pointing out bad style is not enough if the user does not know how to fix it. Fosite therefore uses data-flow analysis to pinpoint the source of problems and uses that information to inform the user.

We would like to be able to recognize suboptimal patterns as well, such as in code sample 13. Replacing all occurrences of this pattern with a call to `min` not only makes the code easier to read, it's also significantly more performant in interpreted languages such as Python. To be

[1] https://github.com/hdelva/fosite/tree/master – accessed 31 May 2017

```
1 if x < y:
2   z = x
3 else:
4   z = y
```

CODE SAMPLE 13: Inefficient Implementations

able to recommend sound code transformations, we will rely on the research done in the area of compiler technology. The analysis should also work as close as possible to the submitted code though, and at the very least maintain a one-to-one relationship to it. This is important because the end goal is automated refactoring, which becomes hard when the input becomes mangled beyond recognition.

3.2 Approach

As pointed out in sections 2.1.1 and 2.3, analysis of dynamic languages is often done using abstract interpreters, and many modern compilers use an SSA form to perform sound optimizations. The PyPy interpreter for Python constructs this form using an abstract interpreter as well [48], although their solution is intraprocedural. Others have successfully used abstract interpreters to perform a may-alias analysis on Python with the goal of optimization [12]. Fosite is based on their conclusions and results, but it ultimately has a different focus, i.e. detailed static analysis and automated refactoring.

The result of our analysis isn't exactly an SSA form for a few reasons. The Memory SSA approach by the GCC and LLVM is a bad fit since it relies on a low-level representation, which is a luxury we don't have as we need to stay close to the original source code. Regular SSA is of course an option since PyPy does it [48], but can't handle collections [27].

Fosite emits more general *use-def* information instead. A *use* refers to any data dependency, such as resolving a variable name or retrieving an element from a collection. A *def* is the opposite such as assigning something to a variable name or inserting an element into a collection. Within Fosite, a *use* and a *def* are respectively called a dependency and a change. As with SSA, incremental numbering can be applied to subsequent definitions. This can be done in a similar fashion as applying the path convergence criterion for the conversion to SSA form. The path-sensitivity of the analysis already provides the necessary components to do so – and control flow is very well-structured in Python (and other dynamic languages). The main difference with regular SSA is that this requires a separate data structure. This allows a single expression or statement to cause multiple changes. A conditional statement is exactly that – a statement, and it can

be useful during refactoring to treat it as a single atomic thing. Its dependencies and changes are the sum of its branches'. In other words, the external data structure allows for a more hierarchical analysis.

To achieve the same level of precision as Memory SSA, Fosite uses two kinds of changes and dependencies. For starters, there's the usual identifier dependency, which is used to model reachability of data. Consider code sample 14 in which `x` gets defined twice. In between assignments, the first value of `x` gets used in a call to `print`. The `print` must come before the second assignment to `x`, as the intended data is no longer reachable after the assignment. Another sort of dependency are object dependencies, which are used to model state dependencies. The second assignment to `x` assigns a list to it, which gets printed as well. Before printing however an element gets appended to it. Appending an element to a list doesn't change anything about the identifier and thus can't be modeled in the same way. In other words, the final call to `print` has a dependency to both the `x` identifier and to whichever object `x` points to at the same time – and both dependencies serve their own purpose.

```
1 x = int(input())
2 y = int(input())
3 print(x - y)
4
5 x = []
6 x.append(1)
7 print(x)
```

Code sample 14: Dependency Example

There are hidden dependencies in this program as well. The first two lines of code will read two lines from `stdin` and parse them to integers. The order in which this happens is important, since the two values get subtracted from each other. This can be modeled by adding implicit state. The call to `input` both depends on the implicit state and changes it, which will ensure that the relative order between the two calls remains the same. Other IO functionality such as the `print` call will do the same. Implicit state can easily be modeled by designating a specific and hardcoded object to be the internal state.

3.3 Languages

Since programming languages tend to share a lot of features, it's not unthinkable to have an abstract interpreter that can process multiple languages. The first thing this would need is a common input format, which in Fosite's case is its *General Abstract Syntax Tree* (GAST). It

has to be able to capture all *syntactic* features of the languages it supports – the semantics aren't relevant for an AST. Adding support for an additional language will require some new nodes or some extra information in existing nodes. Since only things will have to get added interpreting the existing languages can just ignore the additional node types. Another thing that's special about the GAST is that every node has its unique identifier, and the identifiers are totally ordered. If one node's identifier is less than another's, that must mean it came before that other node in the original source file. This is important to accurately report code points, but also because some optimizations rely on it.

The interpreter has to be able to add semantics to the common AST structure. A different `Executor` instance may be assigned for every supported node for every supported language. Languages that share the same features can reuse existing `Executor` implementations. Common and fundamental features such as function scoping are even available inside the interpreter's implementation itself.

As the test data itself is in Python, the Python programming language was the main focus during implementation.

3.4 Analysis

While linters recognize error prone patterns, an interpreter can recognize error prone patterns and logic, as well as some outright errors. An additional benefit of an interpreter-based approach is that it approaches feedback the same way a person would: starting at the beginning, step by step. There are some interesting things that an interpreter can do that linters (or at least PyLint) can't.

```
1 def foo():
2     for ...:
3         for ...:
4             if ...:
5                 return ...
6             elif ...:
7                 return ...
```

Code sample 15: Function Returns

Code sample 15 contains an error prone pattern of Python-like code. A student had written something like this which resulted in one out of 200 unit tests failing. Written like this, it's possible that none of the intended `return` statements are executed. If this happens, the return value is going to be `None`, which makes the unit test fail in an unexpected way – nowhere did

they specify a `None` value should be returned. Fosite gives an accurate description of the cause – the function did not return under these conditions – instead of just the result.

```
x = []
...
while tuple([0] * i) not in x:
    ...
    x += tuple([0] * i)
```

Code sample 16: Heterogeneous Collections

The test data contains an exercise that required that a sequence of tuples should be generated, stopping whenever a tuple of zeroes has been added. Code sample 16 is based on one of the submissions. Up until the addition of the tuple of zeroes, the type of `x` had been `List[Tuple[int]]` (in the notation used by Python 3.5 type hints). Instead of appending the tuple however, `+=` will concatenate the tuple's elements to `x` – in the same way calling `extend` on `x` would. This changes the type to `List[Union[int, Tuple[int]]]`. This transition to a heterogeneous collection is valid Python code but ultimately very error prone. In fact, this causes an infinite loop in this case, as the expected element never gets added.

```
def change(x, d = None):
  list1 = ''
  list2 = []
  for i in range(0, len(x)):
    while x[i] != ' ':
      list2 += x[i]
    list1 += translate(list2[0], d)
    list2 = []
  return list1
```

Code sample 17: Endless Loop

Although deciding whether or not any given program will stop is impossible, it is possible in some cases. Those cases also happen to be quite common. Code sample 17 is an excerpt from a submission. The student intended to tokenize the string `x`, building the token in `list2`. Every token should then get translated and the translated token gets stored in `list1`. There are a number of mistakes, but the most important one is the endless `while` loop. The student wanted index `i` to be a starting position of the token, with the `while` loop building the token from that point. That's of course not what the code does. The same character will get added over and over since none of the values in the loop condition ever change. Data-flow analysis remembers

when and where variables get their values, so it can be used to recognize that the variables are still the same.

The secondary goal of the interpreter is to open the door to code refactoring, through the means of a data-flow analysis. Every evaluated node during interpretation emits a list of changes and dependencies as discussed in section 3.2.

Chapter 4

Implementation

Fosite is an abstract interpreter. It uses abstract pointers, which can be used to fetch abstract objects from an abstract memory. The objects themselves have no value, but they do have a notion of types, attributes, and elements. There is a notion of namespaces where names get mapped to objects, and a notion of heterogeneous collections. In essence, the interpreter tries to get as close as it can to an actual interpreter, without having actual values. This is not as obvious as it sounds. For example, it's tempting to cut corners when implementing the *Method Resolution Order* (MRO), variable capturing in closure definitions, or the explicit `self` argument in Python. Simple approximations of these behaviors would suffice for well written code – but targeting such code makes no sense for a static analyzer. We have to be able to analyze *really* bad code as well.

```
x = 0
y = 0
z = 0

if ...:
    x = 9
    if ...:
        y = 5
        if ...:
            z = 1

a = complex_computation(x)
b = complex_computation(y)
c = complex_computation(z)
```

Code sample 18: State Explosion

As an abstract interpreter uses abstract values, it can't decide which branch to take at every branch point, so it will explore every possible branch. This can quickly lead to a state explosion problem as the number of branches can increase exponentially. Consider the Python-like code in code sample 18. There are branching points on lines 5, 7 and 9, so that the function calls on lines 12, 13, and 14 can each be executed once for each of the four possible execution paths for a total of 12 calls to `complex_computation`. Each argument `x`, `y` or `z` only has two possible values though, so we should be able to do better. The Fosite interpreter will create new execution paths at every branch point, but those paths will also be merged after evaluating the branch point. This means that there will only be a single active execution path left upon executing the calls on lines 12-15. Merging execution paths preserves all the relevant information of each branch, as discussed in section 4.6. The result still has exponential complexity, but no longer in the number of branch points but in the number of possible values

4.1 Objects

As alluded to at the beginning of section 4, the Fosite interpreter keeps track of attributes and elements. Attributes can reuse the namespace logic as described in sections 4.6 and 4.7. Elements are a lot harder to model and are covered in section 4.8.

Everything is an object in Python, even classes become class objects upon definition. An object of a class has a reference to its class object. Among other things, this reference gets used during name resolution. Every class can also extend multiple other classes, called base classes, in a similar way. This can easily be modeled in an abstract interpreter using a list of pointers.

The type of an object is harder to model however. In many object-oriented languages, an object's class is its type. Python's situation is a bit more complex since it has multiple inheritance, and classes are objects as well.

```
1  >>> x = 42
2  >>> t1 = type(x)
3  >>> t1
4  <class 'int'>
5  >>> t2 = type(t1)
6  >>> t2
7  <class 'type'>
8  >>> t3 = type(t2)
9  >>> t3
10 <class 'type'>
```

Code sample 19: Types in Python

Code sample 19 illustrates why this is odd. The `type` function returns a class object, so that `type(42)` returns the class object of name `int`. Using the same function to get the class object's type returns a class object of name `type`. Requesting that object's type reveals something strange – `type` is its own type. This seemingly cyclic dependency gets implemented in CPython using a type flag, if that flag is set it will return the type class object when needed. In other words, the `type` object doesn't have a reference to itself, it will get a reference to itself at runtime when needed.

The type of a value is the same as its class object. A class's basetypes have nothing to do with its type – a class object's type is always `type`. These semantics are quite straightforward to model in an abstract interpreter: the list of base class references are still there, but there's also a type flag. When that flag is set, the `type` function shouldn't use the base classes but fetch the pointer to the `type` class object.

4.2 Paths and Mappings

In order to accurately report the cause of errors and warnings, we need to know the source of every value. A path corresponds to a sequence of code points so that the user gets an idea of the execution path that leads to a problem. A path is an ordered sequence of path nodes. Examples of path nodes include which branch of a conditional was followed, assignments, and function calls. The path nodes should submit a logical ordering, so that users can easily interpret results. For example, this is what a path may look like to the users:

```
Call to numismatist at row 113, column 1
Element of the collection at row 97, column 9
Call to repeater at row 98, column 16
Assignment to lijst at row 113, column 1
Assignment to i at row 97, column 9
Assignment to getal at row 98, column 16
```

As mentioned in section 3.3, AST nodes have totally ordered unique identifiers. A first attempt at defining the path node would be to just reuse the AST identifiers. This works fine until function calls come into the picture. A function call will come after the function definition, and its identifier will be larger than any of the function definition's nodes. This would place the execution of the function body before the function call itself. On top of that, this approach does not support executing the same node more than once. A better solution is to define a path node to be an ordered collection of AST nodes – the nodes that are currently being executed.

Some nodes need extra information. A conditional branch node for example needs to indicate which branch was actually taken. Each branch is incrementally numbered, and contains the total number of branches for practical reasons (see sections 4.6 and 4.10). The actual branch numbers are of no concern. Their main purpose is telling possible branches apart.

Definition 4.1 describes the structure of a path node, and definition 4.2 defines how path nodes are ordered. Note that $\prec$ is the symbol meaning "precedes", and $\prec_{lex}$ means it precedes according to the lexicographic ordering. Definition 4.3 defines a property which will get used in section 4.4 where it will be used to exclude results because they contain paths that would contradict the current execution path.

If we have to merge two paths at any point during execution, we'll need to make sure the paths are mergeable at all. This is important when evaluating binary operations, or during name resolution of an attribute. Definition 4.4 contains the definition of any given node's complementary nodes. Definition 4.5 says that two paths are mergeable if neither path contains nodes that complement any other the other's path nodes. A path's complementary paths can be constructed using the complements of its nodes, this process is described in algorithm 1. Complementary paths will prove useful in section 4.5.

Definition 4.1. A path node is of the form $((n_1, n_2, ..., n_i), b, t)$, where the elements n_i are an ordered sequence of AST node identifiers, b is the number of the branch that was taken, and t is the total of branches that were possible at that node.

Definition 4.2. Let p and q be two path nodes with forms respectively (n_p, b_p, t_p) and (n_q, b_q, b_t), $p \prec q \iff n_p \prec_{lex} n_q \vee (n_p = n_q \wedge b_p \prec b_q)$.

Definition 4.3. A path A is *contained* in another path B if every node of path A occurs in path B as well.

Definition 4.4. The complementary nodes of a single path node (n_p, b_p, t_p) are defined as $\{\, (n_p, i, t_p) \mid 0 \leq i < t \wedge i \neq b_p \,\}$. If $t_p = 1$, an assignment node for example, there are no complementary nodes.

Definition 4.5. A path A is *mergeable* with another path B if a A does not contain a complement of one of B's nodes.

A mapping is simply a pair of the form `(Path, Pointer)`. Because they usually appear in multiples, they can be implemented as a list of `(Path, Pointer)` values instead. In this case, every path in a mapping must be distinct. There are no paths that are contained by another path in the same mapping.

Algorithm 1 Complementary Paths

```
1:  function COMPLEMENT(path)
2:      result ← []
3:      current ← []
4:      for all nodes in path do
5:          if node.is_branch() then
6:              for all complements of node do
7:                  temp = current.clone()
8:                  temp.add_node(complement)
9:                  result ← result ∪ temp
10:         current.add_node(complement)
        return result
```

4.3 Boolean Expressions

A boolean expression can be arbitrarily hard to evaluate. When used in a conditional statement, we can't always decide whether or not a given branch gets taken. The best we can do in these cases is concluding that the branch *might* get taken. Evaluating any boolean expression can thus result in `True` or `False`, as well as `Maybe`.

```
if current is not None:
  print('given {}-{}'.format(current.year, current.month))
else:
  current = datetime.now()
```

CODE SAMPLE 20: Conditions

Code sample 20 shows that in some cases, we really need an accurate answer. This is a pattern that occurs when dealing with optional arguments or when writing library functions. The negative branch should only get executed when `current` was not `None`, so that an actual argument doesn't get overwritten. On the other hand, it *must* be executed if `current` was `None`, so that further evaluation doesn't result in a false type error.

The `is` operator compares the addresses of two objects and returns `True` if and only if they're equal. We can mimic this behavior – and answer with certainty and under which conditions the two operands' point to the same location. The resulting mapping will use the merged paths of the operands to point to the `True` object. The `==` operator should be similar. Technically it depends on the implementation of the `__eq__` method, but let's assume that it has a decent implementation. In that case it should at least return `True` if both operands point to the same object – as with `is`. A similar reasoning can be applied to the `!=`, `<=`, and `>=` operators.

We can also handle the `and`, `or`, and `not` operators in a similar way. If both operands already point to `True`, we can merge the paths and return a mapping that points to `True` as well. The other two operators are analogous.

We combine the paths of both operands to get a new mapping. This means that we must only consider path pairs that are mergeable. Failing to meet this requirement will lead to false positives very quickly.

4.4 Conditionals

Evaluating the test of a conditional branch can give us useful information for the evaluation the individual branches. If that information includes for example that we are sure that `x` is not `None`, we should disregard any mapping that says otherwise. Even better, we can exclude any mapping that would occur under the same contradictory conditions – even if those mappings don't have an explicit connection to `x`.

```
1 if cond1: # Condition 1
2   y = None
3   z = None
4
5 if y is not None: # Condition 2
6   print(z.attribute)
```

Code sample 21: Conditions

In code sample 21, there's an implicit relation between condition 1 and condition 2. Going back to section 4.3, the result of the test of condition 2 will contain a mapping (p, x), where p contains a node indicating that the positive branch of condition 1 was taken, and where x is a pointer value to `False`. This means that any mapping containing p during the execution of the positive branch of condition 2 cannot occur during actual execution. We will call this concept *path exclusion*, and paths such as p are called *restricted paths*. Observation 1 summarizes this more formally.

Observation 1. Assume that resolving an identifier x results in a set of mappings M. Every mapping $m \in M$ is of the form (p, a), where a is a pointer value, and p is the execution path that mapped x to a.

Let R be the set of restricted paths. Given a mapping $(p_m, a) \in M$, if there exists a path p_r in R for which holds that p_m contains p_r (by definition 4.3), we can exclude the mapping from the current evaluation.

4.5 Loops

Without being able to accurately evaluate loop conditions or generators it's impossible to know how many times a loop body gets executed. There are a few different approaches to this problem. The most accurate one is to iterate until we can conclude doing further iterations won't benefit the analysis anymore, we say that a *fixed-point* state has been reached in this case. Every iteration will likely change something though, so that recognizing a fixed-point state isn't as easy as waiting for an iteration that changes nothing.

An easier approach that still has sufficient accuracy is to evaluate every loop body exactly twice. Theoretically it's possible to not even do a single iteration – this leads to false positives however as most exercises can guarantee that some loops will always have to do something. There are two reasons why two iterations is significantly more accurate than a single one. For starters, the first iteration can redefine values that are only used within the loop body. If this redefinition is wrong, this can only be recognized by evaluating it a second time. The second reason is to differentiate between the `break` and `continue` statements. As section 4.6 will discuss, these two statements will hide changes until some later point during execution. For the `break` statement that point is the end of the loop, for the `continue` statement however this point is the beginning of the next iteration.

The analysis of loops goes hand in hand with *watches*, which are used to compare the execution state before and after executing the loop. They start in a setup phase, during which they will learn which components to watch. A new watch gets made at the beginning of evaluating the loop test or loop generator, and it will store all data dependencies (corresponding to those of the data-flow analysis) for that expression. It will contain the returned mappings for the identifiers, along with a list of used objects. The watch leaves the setup phase before evaluating the loop body, and it will now store all data changes for the identifiers and objects that are being watched.

The information in a watch can be used to see whether or not iterating affects the loop test or the loop generator. Knowing under which conditions iteration changed something is easy. The watch already contains that information. Finding the paths that don't change anything isn't as easy because the watches only contain changes – the opposite of what we want. The invariants can be found by taking all the complementary paths of all the changes, and retaining only those that are mergeable with all changes. Section 4.11.2 will discuss how this information can be used as an indicator of error prone code.

4.6 Namespace

Namespaces are the most essential component of the interpreter. We want to give the most descriptive and helpful messages we can, by describing the conditions in which something occurs. Paths are a first step towards describing these conditions, but namespaces are where they're stored. There are a few layers of abstraction required to make this possible, and they will be introduced incrementally.

OptionalMapping

Mappings have already been introduced, but they contain a pointer value which is not enough to indicate an uninitialized variable. A different structure is used to this end, where the pointer value is optional. An `OptionalMapping` with a missing pointer value indicates an uninitialized variable and also describes why the variable is uninitialized.

Branch

The `Branch` struct is the first layer of a namespace and it's where names are added, its internal structure is of the form `HashMap<String, OptionalMapping>`. Every branching point during execution can induce several new branches in a namespace that are separated during execution, so that the negative and the positive branch of a conditional statement do not influence each other for example. A `Branch` struct only contains the changes that happened in its own branch. The changes that happened before branching are stored in different `Branch` structs, which leads to a sparse data structure.

StatisChamber

If we encounter a `break` statement while evaluating a loop body, the evaluation of the current execution path terminates. The changes made until that point have to be saved, as they will become relevant after the loop has been evaluated. Function calls require the same to handle different return points. A `StatisChamber` contains a `HashMap<Path, Branch>`. The path key is used because the control flow can be broken at multiple points.

SubFrame

For every branch point, we use a `Branch` and three `StatisChambers` – two for loops, one for function calls. Loops require two statis chambers to keep the ones caused by a `continue` separate from those caused by a `break`. These four components get stored in a `SubFrame` struct.

Frame

This is the first namespace component that contains some actual logic. Every branch point leads to the creation of a new `Frame`. This structure contains a *cause*, the path node where the branching happened. It also contains a subframe for each possible branch at the cause node. There is only one subframe active at any point during execution and its index is stored. Algorithm 2 describes how a mapping gets inserted into a frame. The `insert` method of a `StatisChamber` will simply insert into each of its branches.

Algorithm 2 Set Mapping

```
1: function SET_MAPPING(name, mapping)
2:     if self.contains(name) then
3:         old_mapping ← self.resolve(name)
4:     else
5:         old_mapping ← OptionalMapping :: new(Path :: empty(), None)
6:     self.current_branch.loop_statis.insert(name, old_mapping)
7:     self.current_branch.function_statis.insert(name, old_mapping)
8:     self.current_branch.branch.insert(name, mapping)
```

Code sample 22 illustrates how the statis chambers are used. When executing the `continue` statement on line 11, the active branch will only contain a mapping for `x` in which it points to an `int` object. This will stop the current execution path, and conclude the execution of the conditional on line 6 as it does not have a negative branch. Rather than merging the contents of subframe `3.1` into subframe `2.1`, it will put the contents into the loop static branch of subframe `3.1`. The key used to store into the statis chamber is a path containing only a single node: the node corresponding to the positive branch of line 8.

The first time the addition on line 19 gets executed is actually safe – `x` and `y` will always have type `str` at this point, and `z` will also receive a mapping to an object of type `str`. Every loop gets evaluated twice as discussed in section 4.5, and a mapping in which `x` points to an `int` since line 10 in the previous iteration is now part of the active branch.

Line 21 isn't safe either. There are at least two execution paths that left `z` uninitialized: if either the condition on line 8 or the one on line 14 was true. Before the addition on line 19 was

```
# frame 1
x = 'x'
y = 'y'

while True:
    if 'cond':
        # frame 2, subframe 2.1
        if 'cond2':
            # frame 3, subframe 3.1
            x = 9
            continue
    else:
        # frame 2,subframe 2
        if 'cond4':
            # frame 4, subframe 4.1
            y = 7
            break

    z = x + y

z + 'z'
```

CODE SAMPLE 22: Broken Control Flow

executed, the previous value was stored into the statis chambers that existed at the time. All but one statis chamber will now contain an empty mapping for `z`, indicating an uninitialized value. These mappings enter the active branch at the end of evaluating the loop, to give us the wanted analysis result.

Namespace

The actual namespace is simply a list of frames, one for each branch point that is being executed. Looking up an identifier is simple: look for the most recent frame that contains a mapping for that name. Name resolution is simple because the other operations do all the heavy lifting. There are three other operations:

- *Grow*: Uses a path to create new frames until there is a frame for each node in the path.
- *Insert*: Grows the namespace when needed and then inserts a mapping into the last frame.
- *Merge*: Merges the last frame into the previous frame.

For the sake of data sparsity, growing is done upon insertion – and not upon the actual branching. Bearing in mind that every object has a namespace as well, we don't want to grow each of

their namespaces every time – its namespaces probably won't even change as most objects are immutable literals.

Grow

If the current namespace has n frames, and the given path has m nodes, we must add $m - n$ frames – corresponding to the last $m - n$ path nodes. The correctness of this approach relies on a bit of inductive reasoning.

The active execution path will always be at least as long as the number of frames in any namespace. All the namespaces that have been changed have the same number of frames. If a change has been made in the current branch, growing has added frames until there are as many frames as there are nodes in the execution path. If a change was made in some branch that is already been executed, and is thus no longer part of the execution path, merge operations will have reduced the number of frames until the length is equal to the length of execution path. The namespaces that have not been changed have strictly less frames, corresponding to the length of the execution path of their last change.

The causal nodes of the frames of any namespace form a prefix of the active execution path. This is a trait of the language. Since Python (or any other language we would target) does not have a `goto` statement, there is a fixed structure to the branching points.

Merge

The merge operation combines the results of the last frame, removes it, and puts the merged result into the frame that is now last. An argument determines the destination of every subframe's content – either into the regular branch or into a statis chamber. The current method is specific to function scoping, but block scoping can be added in a similar way.

```
1 x = 42
2 y = 'string'
3
4 if cond:
5   x = '42'
6
7 x + y
```

Code sample 23: Merging

The name resolution stops at the most recent frame that contains that name. Code sample 23 illustrates that this isn't just an easy solution – it can also be a useful one. If the negative branch

of the condition at line 4 is taken, execution will fail at line 7. Variable `x` still has the value it received at line 1, but only reporting this paints an incomplete picture. It only has that value if the negative branch was taken. So if we want to accurately describe why it has that value, that information should be there as well.

Algorithm 3 Merge

```
1:  function MERGE
2:      names ← [ ]
3:      for all subframes in frames.last() do
4:          names ← names ∪ subframe.names
5:      branch_content ← Branch :: new()
6:      loop_statis ← StatisChamber :: new()
7:      function_statis ← StatisChamber :: new()
8:      cause ← frames.last().cause
9:      for all (i, subframes) in frames.pop().enumerate() do
10:         new_node ← PathNode :: new(cause, i)
11:         subframe.augment_statis_chambers(new_node)
12:         loop_statis.insert_all(subframe.loop_statis)
13:         function_statis.insert_all(subframe.function_statis)
14:         for all name do
15:             mapping ← subframe.resolve(name)
16:             mapping.augment(new_node)
17:             if break_loop(subframe) then
18:                 loop_statis.add(mapping)
19:             else
20:                 if return_function(subframe) then
21:                     function_statis.add(mapping)
22:                 else
23:                     branch_content.add(mapping)
24:     frames.last().insert_all(branch_content)
25:     frames.last().insert_all_loop_statis(loop_statis)
26:     frames.last().insert_all_function_statis(function_statis)
```

Merging begins with collecting the names of all identifiers that have changed in each branch of the frame. A variant of the cause node gets created for every subframe, using the index of the subframe. These augmented nodes are then used to *augment* the paths in the statis chambers, which then get moved to the frame that is now last. The statis chambers are all of the form `HashMap<Path, Branch>`, where the `Path` was supposed to keep the different sources of broken control flow apart. The `augment` method will add a new node to every key path – future additional key paths won't contain this node, which is how different points of broken control flow are being kept separate.

Code sample 23 showed us some behavior we want during name resolution. We can achieve this by incorporating name resolution in the merge operation. Every identifier that has been changed

in any branch will get resolved in every branch, the result gets augmented gets augmented with a variant of the cause node, and is then stored in the frame that is now last. This will ensure that all the relevant mappings for any given identifier can always be found in a single frame. More importantly, the `augment` method will update path mappings to reflect when an identifier didn't change in some particular branch.

Function calls and loops can also create a new frame in namespaces. While merging after a conditional branch can place things inside a statis chamber, merging after a loop or a function call will place the contents of a statis chamber into the active branch. The key paths are merged into the mappings of their corresponding branches first because previous merge operations have not updated those mappings yet.

4.7 Name Resolution

Namespaces by themselves aren't enough to implement all the name resolution behavior; name resolution uses several namespaces. It's possible that a variable is sometimes uninitialized in a namespace, in which case resolution can continue using another namespace. The unresolved paths are carried over to the resolution in the next namespace. All returned mappings have their paths merged with the unresolved paths to reflect the fact that name resolution continued in the next namespace. Algorithm 4 illustrates the general method of name resolution, the `next_namespace` function depends on the kind of name being resolved. An error is emitted if `unresolved` is not empty and there are no more namespaces to try.

Algorithm 4 Resolve

```
1:  function RESOLVE(name)
2:      result ← Mapping :: new()
3:      unresolved ← [ Path :: empty() ]
4:      while unresolved.len() > 0 do
5:          new_unresolved ← [ ]
6:          namespace ← next_namespace()
7:          mapping ← namespace.resolve(name)
8:          for all (p, x) in mapping do
9:              for all unresolved_path in unresolved do
10:                 new_path ← p.merge(unresolved_path)
11:                 if x.is_none() then
12:                     new_unresolved ← new_unresolved ∪ new_path
13:                 else
14:                     result.add(new_path, x)
15:         unresolved ← new_unresolved
16:     return result
```

4.7.1 Identifiers

Identifiers (also called names) are stored in up to four different namespaces:

- The local scope gets created at the start of every function call, which why it's also called the function scope. All variables made during a function call will live here, and stop existing when the function call is done.
- The enclosing scope is most commonly used to capture variables in lambda definitions, but the same principle holds for any sort of function definition. Variables that occur in a function definition and that are already in scope during the definition will get "saved" to the enclosing scope of the function.
- The global scope, where most class and function definitions end up going, and where students first write their first programs in.
- The builtin scope, which is not to be confused with the standard libraries, contains essential Python features such as the `int` type and the `sorted` function.

At any point during execution there are either two or four active scopes, depending on whether or not a function call is being executed. Every function call creates a new empty local scope, but reuses the existing enclosing scope associated with its definition.

4.7.2 Attributes

An object's attributes do not necessarily exist in their own namespace. A lot of them, especially the methods, will exist in the namespace of a class object or one of its base classes. Python uses its own *Method Resolution Order* (MRO) to define the order of the base classes in case of multiple inheritance. The first base class in the definition will get used first, where MRO can be applied again. This means that the second base class of the definition will only get used if the first base class did not contain an attribute of that name, and neither did any of its extended base classes.

4.7.3 Methods

Functions are just callable objects, and a method seems to be a function that's part of an object's namespace. There is one obvious difference however, the explicit self of a method definition isn't part of the method call. This is because Python hides an implicit creation of a `method` object, which isn't just a callable object. In fact, it encapsulates a callable object – the function attribute. It also contains a reference to object it's being called on. The method object itself is callable as

well, and calling it will call the embedded function with the embedded parent object prepended to the arguments.

4.8 Collections

Collections are a challenging component of dynamic programming languages. When an element of a collection is used in any operation, we have to know which one to interpret the result. Static programming languages can just return a value of the defined element type, but dynamic programming languages typically have heterogeneous collections. Features like multiple return values rely on this feature. These are actually tuples, which can be indexed or unpacked just like any other collection. Remembering length and the order of the elements of this tuple is paramount to avoiding false positives.

The solution is similar to namespaces, but with additional abstractions to reflect the additional uncertainty. Not every element gets stored explicitly. If a list contains only objects of type `A`, it doesn't really matter which objects those are. Most collections are iterated over, so that all the elements have to be treated equally, and it doesn't really matter which instance of `A` gets used. One or more *representants* are chosen for each type of object in the collection. The order of elements can be important for heterogeneous collections, which is why these can have multiple representants for the same type of element. Non-empty collection literals such as multiple return values are exceptions. Every element does get stored explicitly for these. Chaining representants together can describe both ordered and unordered collections. Unordered collections do have an order after all, just not a reliable one which programmers should rely on.

The components of a collection are introduced incrementally, in the same way the ones of the namespaces were introduced.

Representants

The core components of a collection are the representants. In essence these are an alias for object pointers. The most important information of a representant is its type, so this has been added as well for the sake of usability.

Chunks

Chunks represent contiguous regions within a collection. Their size is defined as an interval $[a, b]$ with $a \in [0, \infty[\wedge b \in [1, \infty]$. Adding an element to a collection as as part of a loop will

create a chunk with $b = \infty$ as we have no knowledge of how many times a loop gets executed. Chunks also contain representants. For example, if `x` is either an `int` or a `list`, adding it to a collection will add a chunk with two representants.

Chains

A sequence of chunks forms a chain. A notion of length is added here as well. Insertion will either replace an existing element or a new one, so that only the upper bound is affected. The same insertion will however decrement the lower bound of all existing chunks, while simultaneously incrementing their upper bounds. Most of the collection logic is implemented here, such as indexing and slicing.

Branches

Branches form the next layer and they are similar to the ones that are part of namespaces. A branch in the context of a namespace gets initialized as an empty `HashMap<String, Mapping>`, and only stores changes. This sparse structure is possible because with namespaces you know exactly which elements have been changed – they have a unique name. Collections don't have this luxury. Unless we absolutely know which element was changed, we have to assume the entire structure has been changed. This is why a collection's `Branch` contains a list of `(Path, Chain)` tuples, where every `Chain` is an entire "collection" in its own right.

Frames

A collection also has frames, which are similar to the ones in namespaces as well. The main difference is that collections currently don't have a notion of statis chambers. These can certainly be added to improve accuracy, but they are less important for collections as they would only have a noticeable benefit when analyzing heterogeneous collections together with broken control flow. A `Frame` contains a list of branches, the path node that caused the branching, and the index of the active branch.

Collections

Collections are stacks of frames, just like namespaces. They also have a grow and a merge operation, but the implementations are slightly different. Growing still creates a new frame for every branch point, but it will copy the previously active branch's content into every branch of the new frame. Merging is done naively: the last frame gets popped, the paths of its content

gets augmented with the frame's cause node and then replaces the contents of the branch that is now active. A different approach could try to merge branches to the chunk level to avoid data redundancies, but this is a considerable challenge to implement. It might become necessary for large and complex files, but the current solution suffices for now.

4.8.1 Operations

There are a few different ways an element can be added to a collection. The most fundamental way is simply by definition, as a list of mappings. In this case, a single chunk gets added for each mapping – every given element is its own representant in this case for optimal accuracy, and every chunk has a known size of exactly 1. Inserting an element is the most complex operation. Any chunk that has only a single representant of the same type as the new element can have its upper bound incremented. All other chunks have their minimum size decremented – to reflect that one of its elements may have been replaced. A new chunk of size $[0, 1]$ gets wedged in between chunks that haven't had their upper bounds incremented. The most common way to add something to a `list` in Python is through the `append` method, which is a much easier case. We can repeat the previous process, but only applied to the last chunk. We can either increment the upper bound of the existing chunk, or add a new chunk of size exactly 1.

There are a few different ways to retrieve an element from a collection. If done through a dynamically calculated index, a mapping for every representant is returned. We can do better for static indices however. The collections contents get linearized first – replacing the chain of chunks with a list of mappings as described in algorithm 5. This process reveals which possible elements there can be at that position. This is a powerful feature for several reasons. The first (or last) element of a collection can be given some special meaning, even if it's generally a bad idea, in which case retrieving the correct element is a good thing. More importantly, students often use explicit indices instead of unpacking when using multiple return values. A static analyzer might want to recommend using multiple return values instead, but it will only be able to do so if its analysis didn't stumble over the indices. The linearized representation can also be used to create slices of collections, which is a very *pythonic* operation so interpreting it accurately is important.

All indexing operations may return duplicates, because the merge operation isn't very thorough. This can be alleviated by only returning a single mapping for every unique representant. As long as we know that an object had been added, all the different that may have happened are less important.

Algorithm 5 Linearize Collection

```
1: function LINEARIZE(n, chunks)
2:     if chunks.is_empty() then
3:         return []
4:     chunk ← chunks[0]
5:     result ← []
6:
7:     outer:
8:     for (path, repr) in chunk do
9:         acc ← []
10:        for _ in 0 .. chunk.min do
11:            acc.append(Mapping(path, repr))
12:            if acc.len() >= n then
13:                result.append(acc)
14:                continue outer
15:
16:        intermediate ← linearize(n - acc.len(), chunks[1..])
17:        for sequence in intermediate do
18:            result.append(acc ++ intermediate)
19:
20:        for _ in chunk.min .. chunk.max do
21:            acc.append(Mapping :: new(path, repr))
22:            if acc.len() >= n then
23:                result.append(acc)
24:                continue outer
25:
26:            intermediate ← linearize(n - acc.len(), chunks[1..])
27:            for sequence in intermediate do
28:                result.append(acc ++ intermediate)
29:    return result
```

4.9 Function Definitions

A function definition creates a callable object and assigns the result to the given name. The object contains a closure, which in turn contains the function body as a block of AST nodes and the required argument definitions. Calling the closure will map the given arguments to the defined ones, as shown in section 4.10. Once the arguments are in place it executes the function body. Return values get treated as assignments to a `___result` identifier, so the existing namespace logic can be reused. Anonymous functions have a similar implementation.

Python's builtin functions and modules are harder to implement. Depending on the interpreter they may not even be implemented in Python, but in C for example. Although the Fosite interpreter is designed to accommodate multiple languages, it's made with dynamic languages

in mind – C is quite far out of scope. A solution is to implement function behavior in closures, which then get injected into the interpreter as callable objects. These internal functions don't contain a function body AST, but manipulate the interpreter state directly. This is a laborsome endeavor, but it does have a few upsides. Builtin functions such as `sorted` contain hundreds of lines of code – and none of them are relevant to our analysis. Including implementation details in the paths can only confuse users, as these usually have no knowledge of the language internals. Giving a summarized result is both more efficient and more useful.

Modules are implemented as closures that can inject callable objects into the interpreter at runtime. This means that with some time and dedication, third party libraries can be easily added in the same way.

4.10 Function Calls

A few things have to be evaluated before a function call. The call target has to be evaluated first, then the arguments get evaluated from left to right. Evaluating the target will return a mapping which can contain a variable amount of pointers. Evaluating the call target can result in several different function objects, and we have to consider every possible case. These all have to be evaluated independently, which is why a frame can have a variable number of subframes – and why path nodes contain information about how many branches there are.

The call arguments get stored in two collections of mappings: a list for the positional arguments, and a map for the keyword arguments. All these arguments get augmented with a path node of the current function call. These then get mapped to arguments in the definition. Python has four kinds of arguments in a function definition: `args`, `kwonlyargs`, `vararg`, and `kwarg`. Both the `args` and `kwonlyargs` can be given default values, and all arguments after the `varargs` will be placed in `kwonlyargs`. The underlying semantics are less trivial than one might expect. The principle is illustrated in algorithm 6. The algorithm assumes every argument is only provided once – the Python runtime already gives a detailed error when this isn't the case.

The calling mechanism is best explained in pseudocode as well, as in algorithm 7. Lines 2 to 5 move existing enclosing and local scopes to a different stack. This makes name resolution easier, as there are always at most four active scopes. Lines 7-9 retrieve the existing enclosing scope, a local scope is created, and both get placed on the stack of active scopes. Lines 11 and 12 contain the function call. A callable object in the interpreter gets retrieved, and is executed using the provided arguments. The callable will do the argument assignments as in algorithm 6 before executing whatever function logic it contains. Line 14 pops the local scope, and retrieves the entry that holds the return value. Lines 15-18 restore the scopes to the state before the function

Algorithm 6 Assign Arguments

```
1: function ASSIGN__POS(gpos, gkw, arg, kwonly, vararg, kwarg)
2:     if gpos.len() > 0 && arg.len() > 0 then
3:         assign(arg[0].name, gpos[0])
4:         assign_pos(gpos[1..], gkw, arg[1..], kwonly, vararg, kwarg)
5:     else
6:         assign_vararg(gpos, gkw, arg, kwonly, vararg, kwarg)
7:
8: function ASSIGN__VARARG(gpos, gkw, arg, kwonly, vararg, kwarg)
9:     if vararg.is_some() then
10:        arg_list ← abstract_list(gpos)
11:        assign(vararg.name, arg_list)
12:    assign_kw(gkw, arg ++ kwonly, kwarg)
13:
14: function ASSIGN__KW(gkw, arg, kwarg)
15:    if (arg.names ∩ gkw.names).len() > 0 then
16:        name ← (arg.names ∩ gkw.names).pick_one()
17:        assign(name, gkw[name])
18:        assign_kw(gkw \ name, arg \ name, kwarg)
19:    else
20:        assign_kwarg(gkw, arg, kwarg)
21:
22: function ASSIGN__KWARG(gkw, arg, kwarg)
23:    if kwarg.is_some()  then
24:        arg_dict ← abstract_dict(gkw)
25:        assign(kwarg.name, arg_dict)
26:    assign_arg_default(arg)
27:
28: function ASSIGN__ARG__DEFAULT(arg)
29:    if arg.len() > 0 && arg[0].has_default() then
30:        assign(arg[0].name, arg[0].default)
31:        assign_arg_default(arg[1..])
```

call. Algorithm 7 does not contain some necessary bookkeeping operations. The local scope gets discarded after handling each call target, but the namespaces of the changed objects have to be merged after handling all call targets, and the resulting mapping needs to have its paths augmented with the path to get to the target.

Interpreting recursive functions requires a way to *tie the knot* – so that analysis doesn't get stuck in an endless loop. The interpreter maintains a call stack, and only permits the same function to be called a set amount of times. An execution branch that would perform a recursive call that exceeds the recursion depth is simply terminated, so that only the branches that result in a base case get executed at this depth. A call at a higher recursion depth will then use this

Algorithm 7 Function Calls

```
1: function CALL(target, args, kwargs)
2:     b ← scopes.len() > 2
3:     if b then
4:         shadow_scopes.push(scopes.pop())
5:         shadow_scopes.push(scopes.pop())
6:
7:     enclosing ← get_closure(target)
8:     scopes.push(enclosing)
9:     scopes.push(Scope :: new())
10:
11:    fun ← get_callable(target)
12:    fun(args, kwargs)
13:
14:    result ← scopes.pop().extract_result()
15:    scopes.pop()
16:    if b then
17:        scopes.push(shadow_scopes.pop())
18:        scopes.push(shadow_scopes.pop())
19:    return result
```

value in its analysis of the entire function body. This method's accuracy is sufficient for most use cases, even for low recursion depth limits [12].

4.11 Warnings and Errors

The interpreter itself can recognize some interesting things. If the interpreter encounters something it cannot process, the actual runtimes probably won't be able to either. In this case the interpreter will emit an error, but with more information about the conditions that lead to the error. The result is similar to printing a call stack, but with even more detail.

It might also encounter error prone patterns during execution. If execution does fail, the warnings' information will add to the errors' to paint an even more accurate picture. Even if execution doesn't fail, some of the warnings might indicate a deeper logical flaw.

4.11.1 Errors

Python doesn't always validate the input to the builtin functions. The `sum` function will just error with `TypeError: unsupported operand type(s) for +` if the argument wasn't an iterable containing things that can be summed together. We can be more helpful in our errors by describing which elements can't be added together and how they got there, perhaps even hinting

towards implementing `__add__` and `__radd__`. Similar errors can be emitted when trying to incorrectly apply an operator to an object, such as adding an `int` and a `str` or trying to insert an element into a `str`.

Uninitialized variables are some of the most common bugs in many programming languages. Badly written code can make these hard to track down. The way our interpreter is structured gives us all the needed information to provide helpful errors.

Every branch of a collection knows its own largest possible length. This length can be used to compare to statically known indices, to make sure there are at least enough elements in the collection. This can be quite useful as students often try to get the first or last element out of a empty list.

4.11.2 Warnings

A variable can point to objects of different types. With the exception of the numeric types, this makes the variable very unwieldy to use. This can be checked after merging a namespace by simply looking at the contents, and looking up the types of the results.

Heterogeneous collections are hard to deal with; unless there's a fixed structure to the collection, operations on its elements have to support all of the possible types. A collection keeps track of its elements' types. Adding something new to a collection will compare the types before and after adding, and will emit a warning if something's changed.

Changing a collection while iterating over its contents is generally a bad idea. Python will not throw an error when this happens, which can lead to weird behavior such as skipping elements, endless loops, and collections consuming all available memory. This can be checked using the watches introduced in section 4.5.

A while loop whose condition does not change after an iteration is prone to endless loops. This is the case if all the variables in the condition still point to the same object, and all those objects haven't been changed either. This can be done using watches as well.

A function call that did not end with an explicit return statement will return a `None` value. This can lead to confusing errors when a user forgot to return in just one of its branches. Since return values are treated as regular identifiers, we can use the existing logic provided by the namespaces to see which `OptionalMapping`s are still uninitialized at the end of a function call.

Chapter 5

Results and Discussion

The ultimate goal was helping students learn faster, but testing this would ideally use a semester-long A/B test. This is quite unethical in a educational context, so it's not an option. We can analyze old submissions, and the Dodona platform has over 800,000 of them, but the results would need manual verification to weed out false positives. In fact, most students work incrementally, implementing the required functionality piece by piece. Pointing out that they haven't implemented some functions is useless, since they were aware of that already.

Seven interesting submissions from three different students have been manually selected instead. That may be a small selection of students, but even helping a single student overcome the initial hurdles of learning to program is a worthwhile effort. Some of the submissions stood out because evaluating them was terminated because the time limited was exceeded. This can mean two things: the submission was too inefficient, or there's an endless loop in the implementation. Students have expressed that they want different error messages for the two cases, and just referring them to the halting problem isn't very helpful. The next best thing is recognizing patterns that could lead to endless loops, which is exactly what the Fosite analysis tool is able to do.

As the submissions tend to be large and hard to read, a prior section will show analysis results on handcrafted trivial examples. These examples will also contain the results of the data-flow analysis, as these results are too hard to interpret (and represent) for actual submissions.

5.1 Artificial Examples

The analysis result contains four things: the analyzed code, the data dependencies, the data changes, and the feedback that's given during interpretation. There are two sorts of entries in the data-flow results as introduced in section 3.2: identifiers and object pointers. Every line contains all the changes or dependencies of that line, in no particular order.

The results of some statements are hard to express on a single line. Analysing a conditional statement is done in multiple steps – the test gets evaluated before evaluating the two branches. These steps are separated in the data-flow representation using a `|` character and indentation. Everything left of the `|` is part of the test, everything to the right is part of the body. As mentioned in section 3.2, the result is hierarchical; the sum of all changes and dependencies of a conditional body is placed to the right of the `|`. Functions are expressed in a similar way. Everything to the left of the `|` corresponds to the definition of the function. Its only change is usually the function name it introduces, but it may have dependencies to any variables the definition might capture as described in 4.9. The right hand side of the `|` corresponds to the changes and dependencies of a function call. The function call itself copies the sum of those results, with the exception that all identifier information is discarded as they're only relevant within that function's scope.

To keep the examples small, placeholder strings are used instead of actual conditions. Non-empty strings always evaluate to `True` in Python, but the analyzer currently evaluates all strings to `Maybe`. Builtin objects in Python can't be given any new attributes either, but it's more convenient to allow this in the analyzer – for now at least – as well.

5.1.1 Assignments

code

```
1 y = 42
2
3 if 'cond1':
4   y.attr = 7
5   y = 6
6
7 y.attr
```

dependencies

```
-

- | y
    y
    -

y, y.attr, 36, 40
```

changes

```
y

- | y, y.attr, 36
    y.attr, 36
    y

-
```

analysis

```
Error at row 7, column 1
 Object y does not have an attribute attr
 In the following cases:
 Case 1
  Assignment to y at row 1, column 1
  Condition at row 3, column 1 is false

 Case 2
  Condition at row 3, column 1 is true
  Assignment to y at row 5, column 2
```

Listing 1: Assign

Listing 1 illustrates how assignments are analyzed. Line 4 resolves the name `y` to some object, and assigns `7` to the attribute with name `attr` in the object's namespace. The changes show that `y` points to the object at address 36. Line 7 resolves `y` as well, and `attr` is resolved in the resulting objects' namespaces. We can see that line 5 created a new object at address 40. Not shown here are the objects 37-39, which are the `'cond1'` string, its boolean value, and the `7` that is used on line 4. Line 7 depends on both the `y` and `y.attr` identifiers, as well as the two possible objects `y` might refer to. This might seem like duplicate information, but two different identifiers can point to the same object, and can thus change its internal state as well as shown in listing 2.

The analysis reveals that `y.attr` will never exist upon executing line 7 as the object that does have an `attr` attributed is no longer reachable.

5.1.2 Aliasing

code

```
1  y = 5
2  x = y
3
4  if 'cond':
5    y.attr = 9
6
7  x.attr
```

dependencies

```
-
y

- | y
    y

x, x.attr, 36
```

changes

```
y
x

- | y.attr, 36
    y.attr, 36

-
```

analysis

```
Error at row 7, column 1
  Object x does not have an attribute attr
  In the following cases:
  Case 1
    Assignment to y at row 1, column 1
    Assignment to x at row 2, column 1
    Condition at row 4, column 1 is false
```

Listing 2: Aliasing

Both `x` and `y` point to the same object in listing 2, so that the resolution of `x.attr` on line 7 depends on the assignment to `y.attr` on line 5. Both `x` and `y` point to object 36, but the paths in the mappings are different – the mapping for `x` also has a path node for the assignment to `y`. The error message on line 7 presents this aliasing information in a human-readable form.

5.1.3 Call Clobbering

code

```
1  y = 9
2
3  def foo(x):
4      x.attr = 'changed'
5      return 42
6
7  if foo(y):
8      print(y)
```

dependencies

```
-

- | x
    x
    -

y, foo | y, print
         y, print
```

changes

```
y

foo | x.attr, 36
      x.attr, 36
      -

36 | 5
     5
```

Listing 3: Clobbering

Section 3.2 mentioned that we'd like an interprocedural analysis, unlike the intraprocedural analysis that's part of PyPy. Call clobbering is one of the challenges to achieve this. The test

on line 7 in listing 3 calls `foo` with argument `y`. Line 4 assigns `'changed'` to attribute `attr` of the given argument. The call to `foo` has side-effects, which can be seen in the data-flow analysis on lines 3 and 7. It's the test of the conditional on line 7 that has the side-effects, so there is a change to the left of the `|`. The `print` function itself has side-effects as well, which gets represented using an internal state object – which happens to be on position 5.

5.1.4 Path Exclusion

code

```
1 def foo(x=None):
2   x + [1]
3
4   if x is None:
5     x = []
6
7   x + [1]
8
9 foo()
```

dependencies

```
- | x
    x

    x | -
        -

    x

foo
```

changes

```
- | x
    -

    - | x
        x

    -

-
```

Analysis

```
Error at row 2, column 2
  Incompatible types for operation +
  The following combinations exist:
  Combination 0: NoneType + list[int]
    Left side has type NoneType
    In the following cases:
      Call to foo at row 9, column 1
      Assignment to x at row 9, column 1

    Right side has type list[int]
    In the following cases:
      Always
```

Listing 4: Path Exclusion

Section 4.4 introduced the concept of path exclusion as a means to support simple type checks and other forms of input validation. Listing 4 gives an example of this principle in action. The definition of `foo` includes an optional argument `x`. This argument should probably be a list, in order to evaluate lines 2 and 7. Default values should be immutable in Python because subsequent calls to the same function use the same default values. The usual solution is to have a default value of `None`, and to initialize it at the start of the function with a fresh empty list. The list concatenation on line 2 occurs before this initialization, so it results in an error. The

one on line 7 is fine however, and the analysis recognizes it as such. Without path exclusion there would be two possible mappings for `x` after merging on line 4 – but as the path where `x` is `None` cannot occur in the negative branch of the conditional, that mapping does not find its way back into the scope after merging.

5.1.5 Control Flow

code

```
1 while 'cond':
2   if 'other cond':
3     x = 'string'
4     break
5   x = 42
6   y = x + 9
7
8 y + 4
9 x + 9
```

dependencies

```
- | x
    - | -
        -
        -
    -
    x

y
x
```

changes

```
- | x, y
    - | x
        x
        -
    x
    y

-
-
```

analysis

```
Error at row 8, column 1
  y does not exist
  In the following cases:
  Case 1
    Condition at row 2, column 3 is true

Error at row 9, column 1
  Incompatible types for operation +
  The following combinations exist:
  Combination 0: string + int
    Left side has type string
    In the following cases:
      Assignment to x at row 3, column 5

    Right side has type int
    In the following cases:
      Always
```

Listing 5: Control Flow

Breaking out of a loop hides all the changes made in that branch for the duration of the loop, but those changes become visible again after the loop. Listing 5 contains an example where this is important. The conditional on line 2 determines the type of `x` after the loop, but is completely irrelevant in the loop itself. Line 9 might cause an error if the condition on line 2

was true. More interesting is that it also finds uninitialized variables, as explained in the statis chamber subsection of section 4.6.

5.1.6 Endless Loops

code

```
1  def square(x, n):
2    if n == 0:
3      return 1
4
5    y = 1
6
7    while n > 1:
8      if n % 2 == 0:
9        x = x ** 2
10       #n //= 2
11     else:
12       y = x * y
13       x = x ** 2
14       n = (n-1) // 2
15
16   return x * y
17
18 square(500, 30)
```

dependencies

```
- | x, y, n
    n | -
        -

    -

    n | x, y, n
        n | x
            x
            -
        - | x, y, n
            x, y
            x
            n

    x, y

  square
```

changes

```
- | x, y, n
    - | -
        -

    y

    - | x, y, n
        - | x
            x
            -
        - | x, y, n
            y
            x
            n

    -

  -
```

Analysis

```
Warning at row 7, column 2
  Not all code paths update the loop condition
  There's a risk of endless loops
  In the following cases:
  Case 1
    Condition at row 8, column 3 is true
```

Listing 6: Endless Loop

Listing 6 contains an example of the most trivial cause of endless loops: forgetting to update the loop condition. The code itself is a rough implementation of the exponentiation by squaring algorithm. It computes x^n, and uses the binary representation of `n`. This is done using bitwise shifts (or division by 2) until there are no more bits left. This halting criterion translates to the condition on line 7. If the condition on line 8 is true however, that iteration of the loop will not even affect the loop condition – a clear case of an endless loop. The invariance of the loop condition is detected using the watches introduced in section 4.5.

code

```
1  a = [1, 2, 3, 4]
2
3  while 0 not in a:
4    if 'cond1':
5      if 'cond2':
6        pass
7      else:
8        a[0] = '0'
9    else:
10     a[1] = 9
```

dependencies

```
-

a | -
    - | -
        - | -
            -
        - | a
            a
    - | a
        a
```

changes

```
a

- | 36
    - | -
        - | -
            -
        - | 36
            36
    - | 36
        36
```

Analysis

```
Warning at row 8, column 4
  Adding an element of a new type to a collection
  a had type list[int] and became list[int, string]
  This can make working with this collection more difficult

Warning at row 3, column 1
  Not all code paths update the loop condition
  There's a risk of endless loops
  In the following cases:
  Case 1
    Condition at row 4, column 2 is true
    Condition at row 5, column 3 is true
```

LISTING 7: Endless Loop

The loop in listing 7 is harder to detect. The loop condition will often will depend on some internal state of an object, rather than a variable mapping. There is a code path that does not alter the variable mapping nor the internal object state, which results in a warning. The other two execution paths aren't very helpful either though, even if they do alter the object state. This is where the invariance checks meet their limits. Other aspects of the analysis may still prove useful however, such as the type warning on line 8.

While loops are often discouraged in favor of for loops, as the latter is less prone to implementation oversights. That doesn't mean they're entirely safe though, as the code in listing 8 illustrates. The code in the example will consume all available memory until the operating system intervenes. There's also a subtle difference between the augmented assign `x=` and the `+` operator when applied to lists. The latter creates a new list by concatenating its two operands, while the former will extend the `lvalue` of the assignment. Python does not crash when changing a collection that's being iterated over.

code

```
1 x = [1]
2
3 for i in x:
4   x *= 2
```

dependencies

```
-

x | x
    x
```

changes

```
x

i | x
    x
```

Analysis

```
Warning at row 3, column 1
  Some code paths change the collection that's being iterated over
  This can have unexpected consequences
  In the following cases:
  Case 1
    Assignment to x at row 4, column 2
```

LISTING 8: Endless Loop

5.1.7 Static Indexing

As discussed in section 4.8, static indexes enjoy special treatment. Listing 9 contains a possible type error on line 11. The second element of `x` gets added to the third element, but the second element is either an `int` or a `str` depending on the condition on line 1. The error message accurately describes this problem. The analysis did not suffer any loss of accuracy because of the collection.

Notice how lines 8 and 9 depend on the identifier `foo` as well as the state of object `41`. The identifier dependency indicates which object the identifier should point to, the object dependencies indicate that those lines depend on the original contents of the collection – as there have been no changes to object 41 yet.

Out-of-Bounds Index

Section 4.8 discussed how every branch in the `Collection` struct contains its possible length as a size range. The upper bound can be used to compare to static indexes to detect out of bounds errors. Line 7 of listing 10 tries to get the third element of `x`, but if `x` has the value it has been assigned to on line 2 it only has two values. Line 6 succeeds because `x` has either two or three elements, so getting the second one is safe. Lines 6 and 7 both depend on identifier `x`, as well as the two possible objects it might point to.

code

```
1  if 'cond':
2    x = 2
3  else:
4    x = 'x'
5
6  foo = [1, x, 3, 4]
7
8  y = foo[1]
9  z = foo[2]
10
11 y + z
```

dependencies

```
- | -
    -
- | -
    -

x

foo, 41
foo, 41

y, z
```

changes

```
- | x
    x
- | x
    x

foo

y
z

-
```

Analysis

```
Error at row 11, column 1
  Incompatible types for operation +
  The following combinations exist:
  Combination 0: string + int
    Left side has type string
    In the following cases:
      Condition at row 1, column 1 is false
      Assignment to x at row 4, column 2
      Assignment to foo at row 6, column 1
      Element of the collection at row 8, column 5
      Assignment to y at row 8, column 1

    Right side has type int
    In the following cases:
      Assignment to foo at row 6, column 1
      Element of the collection at row 9, column 5
      Assignment to z at row 9, column 1
```

Listing 9: Static Index

code

```
1  if 'cond':
2    x = (1,2)
3  else:
4    x = (1,2,3)
5
6  x[1]
7  x[2]
```

dependencies

```
- | -
    -
- | -
    -

x, 38, 41
x, 38, 41
```

changes

```
- | x
    x
- | x
    x

-
-
```

Analysis

```
Warning at row 7, column 1
  Index might be out of bounds
  x does not always have enough elements
  In the following cases:
  Case 1
    x has at most 2 elements in the following case
    Condition at row 1, column 1 is true
    Assignment to x at row 2, column 2
```

Listing 10: Static Index

Unpacking and Slicing

Slicing and unpacking collections are both closely related to indexing with static indexes. Listing 11 contains an example with slicing and unpacking. Everything except for the first and last element of `foo` gets placed in a list called `b` on line 8. Line 9 then unpacks the two values and assigns to identifiers `d` and `e`. The analysis also recognizes when these operations will certainly fail, as they reuse the same logic as static indexing.

code

```
1  if 'cond':
2    x = 'str'
3  else:
4    x = 2
5
6  y = [1, x]
7  foo = y + [3, '4']
8  a, *b, c = foo
9  d, e = b
10 d + e
```

dependencies

```
- | -
    -
- | -
    -

x
y
foo, 47
b, 48
d, e
```

changes

```
- | x
    x
- | x
    x

y
foo
a, b, c
d, e
-
```

Analysis

```
Error at row 10, column 1
  Incompatible types for operation +
  The following combinations exist:
  Combination 0: string + int
    Left side has type string
    In the following cases:
      Condition at row 1, column 1 is true
      Assignment to x at row 2, column 2
      Assignment to y at row 6, column 1
      Assignment to foo at row 7, column 1
      Assignment to b at row 8, column 1
      Element of the collection at row 9, column 1
      Assignment to d at row 9, column 1

    Right side has type int
    In the following cases:
      Assignment to y at row 6, column 1
      Assignment to foo at row 7, column 1
      Assignment to b at row 8, column 1
      Element of the collection at row 9, column 1
      Assignment to e at row 9, column 1

      Assignment to y at row 6, column 1
      Assignment to foo at row 7, column 1
      Assignment to b at row 8, column 1
      Element of the collection at row 9, column 1
      Assignment to e at row 9, column 1
```

Listing 11: Unpacking

5.2 Submissions

Submission 1

FIGURE 5.1: Submision 1

(A) Code

```
from cmath import cos, tan
getal = list(input())
x = 0
y = 0
for i in range(0, 100):
    x += float(1 / float(cos(getal[i] * 36)))
    y += float(tan(getal[i] * 36)) * x
print('{}, {}'.format(x, y))
```

(B) Analysis

```
Error at row 6, column 26
  Invalid argument type
  The first Argument should have one of the following types:
      ["number"]
  It has an invalid type in the following cases:
    Type str in the following case:
      Call to input at row 2, column 14
      Call to list at row 2, column 9
      Assignment to getal at row 2, column 1
      Element of the collection at row 6, column 30
      Call to cos at row 6, column 26
```

```
Error at row 7, column 16
  Invalid argument type
  The first Argument should have one of the following types:
      ["number"]
  It has an invalid type in the following cases:
    Type str in the following case:
      Call to input at row 2, column 14
      Call to list at row 2, column 9
      Assignment to getal at row 2, column 1
      Element of the collection at row 7, column 20
      Call to tan at row 7, column 16
```

Figure 5.1 contains a submission to one of the first assignments of the semester – it's one of their first pieces of Python code. The problem in this submission is obvious, line 2 assigns a list of strings to `getal`, but its elements get used as an arguments to `cos` and `tan`. For new programmers, it can take some time to get used to the fact that not everything that looks like a number is actually a number. Python's error message isn't very helpful: `TypeError: a float is required`. It doesn't even mention *where* the float is required. This student submitted the same code multiple times, and they even tried adding explicit float casts to remedy the error message. Perhaps the student wasn't aware that you can multiply a string with an integer in Python to duplicate the string, and ruled out the possibility that the `getal[i]` was to blame. In any case, Fosite provides the user with additional information to help with debugging.

Submission 2

FIGURE 5.2: Submision 2

(A) Code

```
lijst = str(input())
start = int(input())
sprong = int(input())
sprong2 = sprong
woord = lijst[start]
lijst2 = str(woord)
x = len(lijst) - 1
a = len(lijst) - abs(start)
while x != 0:
    if lijst == 'say hello to a good buy':
        lijst2 += 'y luaeb h o dtyo aoosgl'
    else:
        x -= 1
        d = abs(sprong2)
        if d >= a:
            if sprong2 < 0:
                sprong2 += len(lijst)
            else:
                sprong2 -= len(lijst)
            y = start + sprong2
            woord2 = lijst[y]
            lijst2 += str(woord2)
            sprong2 += sprong
        else:
            y = start + sprong2
            woord2 = lijst[y]
            lijst2 += str(woord2)
            sprong2 += sprong
print('{}'.format(lijst2))
```

(B) Analysis

```
Warning at row 9, column 1
  Not all code paths update the loop condition
  There's a risk of endless loops
  In the following cases:
  Case 1
    Condition at row 10, column 5 is true
```

Students sometimes shoot themselves in the foot such as in fig. 5.2. The suspicious check on line 10 is a case of a student trying to game the system by hardcoding the result for one of the unit tests. They probably didn't expect that this would cause an endless loop though. Students don't usually start working around the unit tests until they've given up out of frustration, at which point an endless loop won't exactly help their mental state. The analysis in this case highlights the endless loop, but not the original cause of their frustration.

Submission 3

FIGURE 5.3: Submision 3

(A) Code

```
hoeveel = int(input())
code = str(input())
signaal = ''
while hoeveel != 0:
    for i in range(0, len(code)):
        while code[i] == ' ' or code[i] == '1' or code[i] == '2' or code[i] == '3'
        ↪  or code[i] == '4':
            signaal += '.'
        if code[i] == '6' or code[i] == '5' or code[i] == '7' or code[i] == '8' or
        ↪  code[i] == '9':
            if code[i + 1] == ' ' or code[i + 1] == '6' or code[i + 1] == '5' or
            ↪  code[i + 1] == '7' or code[i + 1] == '8' or code[i] == '9':
                signaal += code[i]
            else :
                while code[i + 1] != code[i + 1] == ' ' or code[i + 1] == '1' or
                ↪  code[i + 1] == '2' or code[i + 1] == '3' or code[i + 1] == '4'
                ↪  or code[i + 1] == '5' or code[i + 1] == '6' or code[i + 1] ==
                ↪  '7' or code[i + 1] == '8' or code[i + 1] == '9':
                    if code[i + 1] == code[i + 1].upper:
                        signaal += code[i]
    print('{}'.format(signaal))
    signaal = ''
    hoeveel -= 1
    code = str(input())
```

(B) Analysis

```
Error at row 13, column 24
  Incompatible types for operation ==
  The following combinations exist:
  Combination 0: str == method
    Left side has type str
    In the following cases:
      Call to str at row 2, column 8
      Assignment to code at row 2, column 1
      Element of the collection at row 13, column 24

    Right side has type method
    In the following cases:
      Call to str at row 2, column 8
      Assignment to code at row 2, column 1
      Element of the collection at row 13, column 39
```

```
Warning at row 6, column 9
  Not all code paths update the loop condition
  There's a risk of endless loops
  In the following cases:
    Always
```

```
Warning at row 12, column 17
  Not all code paths update the loop condition
  There's a risk of endless loops
  In the following cases:
    Always
```

It happens quite often that a student wants to check some condition in the body of a loop, but uses a `while` loop instead of a simple `if` statement. This makes some sense since the condition should indeed get checked multiple times – but that's what the outer loop is for. Figure 5.3 contains a submission where a student did exactly this. The loops on lines 6 and 12 were obviously not meant to be while loops. Oddly enough the checks on line 8 and 9 are fine, which could mean that the student is close to realizing which language construct was the right choice.

Somewhere in the barely legible code, there's a more subtle mistake. The `==` operator can compare objects of any type. In the case of the condition on line 13, this causes the check to be trivially false – a string is being compared to a method. The student simply forgot to call the method, which is a mistake that easily goes unnoticed. Fosite detects this, and gives an outright error that two incompatible types are being compared. This makes Fosite more strict than Python, in the same way that Google's Error Prone analyzer is more strict than Java (see code sample 1).

Submission 4

FIGURE 5.4: Submision 4

(A) Code

```python
def volgende(reeks):
    volgende_reeks = []
    for i in range(0, len(reeks)):
        volgende_reeks.append(abs(reeks[i] - reeks[(i+1)%len(reeks)]))

    return tuple(volgende_reeks)

def ducci(reeks):
    ducci = []
    ducci.append(tuple(reeks))
    volgend = volgende(reeks)
    while volgend not in ducci:
        ducci += volgend
        volgend = volgende(volgend)
    if volgend != tuple([0]*len(reeks)):
        ducci += volgend

    return tuple(ducci)

ducci([32, 9, 14, 3])
```

(B) Analysis

```
Error at row 13, column 9
  Incompatible types for operation +
  The following combinations exist:
  Combination 0: list[tuple] + tuple[int]
    Left side has type list[tuple]
    In the following cases:
      Assignment to ducci at row 9, column 5

    Right side has type tuple[int]
    In the following cases:
      Assignment to volgende at row 1, column 1
      Call to ducci at row 20, column 1
      Call to volgende at row 11, column 15
      Call to tuple at row 6, column 12
      Assignment to volgend at row 11, column 5
```

Some endless loops are caused by type errors such as in fig. 5.4. Using the syntax introduced in Python 3.5 for type hints, the `volgende` function returns an object of type `Tuple[int]`. The loop that starts on line 12 should add additional `Tuple[int]` values to the collection with name `ducci` until the value that would get added is already part of the collection. Line 13 does not add `Tuple[int]` values to the collection though – it adds every `int` separately. Using the augmented `+=` assignment operator on a list is equivalent to calling the `extend` method – and not the `append` method that the student should have used. No `Tuple[int]` values get added, and the loop never terminates. The endless loop itself is hard to recognize, but Fosite does warn when addings elements of a new type to a collection. Personal correspondance has shown that giving this warning along with explaining the difference between appending to a collection and extending a collection would've been enough for the student to fix their mistake themselves.

Submission 5

FIGURE 5.5: Submision 5

(A) Code

```
1 def letterwaarde(planeet):
2     planeet1 = planeet
3     while '-' or '@' in planeet1:
4         planeet = ''
5         for x in planeet1:
6             if x.isalpha():
7                 planeet += x
8
9 letterwaarde('EARTH')
```

(B) Analysis

```
Error at row 3, column 11
  Incompatible types for operation or
  The following combinations exist:
  Combination 0: str or bool
    Left side has type str
    In the following cases:
      Always

    Right side has type bool
    In the following cases:
      Call to letterwaarde at row 9, column 1
      Assignment to planeet at row 9, column 1
      Assignment to planeet1 at row 2, column 5
```

```
Warning at row 3, column 5
  Not all code paths update the loop condition
  There's a risk of endless loops
  In the following cases:
    Always
```

Executing `import this` in Python prints out "The Zen of Python" which is an informal Python style guide. Number two on that list is "Explicit is better than implicit", but oddly enough truthiness is considered to be *pythonic*. The truthiness of a value is its boolean value, and in practice this is equivalent to an implicit `bool` cast. This is quite error prone as fig. 5.5 illustrates. The condition on line 3 is always true, since `'-'` is never empty. Fosite does not consider `str` values to be usable in boolean operations. They can used as standalone boolean expressions

however, so that truthiness can still be used for input validation, which is the primary use case for truthiness.

Submission 6

FIGURE 5.6: Submision 6

(A) Code

```
warmedag=0
hetedag=0
x=input()

while x!= 'stop':
    x=float(x)

    for x in range(0,5):
        while x >= 25.0:
            warmedag+=1
            x=input()

        while x>=30.0:
            hetedag+=1
            x=input()
        else:
            hetedag=0
            warmedag=0

if warmedag==2 and hetedag==3:
    print('hittegolf')
else:
    print('geen hittegolf')
```

(B) Analysis

```
Error at row 13, column 15
  Incompatible types for operation >=
  The following combinations exist:
  Combination 0: str >= float
    Left side has type str
    In the following cases:
      Iteration of the loop at row 5, column 1
      Iteration of the loop at row 8, column 5
      Iteration of the loop at row 9, column 9
      Call to input at row 11, column 15
      Assignment to x at row 11, column 13

    Right side has type float
    In the following cases:
      Always
```

Reusing the same identifiers can have unexpected consequences, such as in fig. 5.6. The `for` loop on line 8 redefines `x`, while the condition on line 9 was intended to use the old value. This actually means that the test on line 9 immediatly returns false, so that the loop is never executed, and the same thing happens to the next loop. In effect, the iterations of the loop on line 5 don't change anything. Unfortunately Fosite doesn't know that the tests on line 9 and 13 are trivially false. This means that it doesn't find the most urgent problem, but it does find another problem – line 11 should have a `float` cast.

Submission 7

Figure 5.7: Submision 7

(a) Code

```
25 def radar(getal):
26     if standvastig(getal) == True:
27         return False
28     getal = str(getal)
29     getal = list(getal)
30     if len(getal) // 2 == len(getal) / 2:
31         a = len(getal) // 2
32         n = 0
33         for z in range(0, len(getal) // 2):
34             if getal[0] == getal[-1]:
35                 n += 1
36                 getal.reverse()
37                 getal = getal[:-1]
38                 getal.reverse()
39                 getal = getal[:-1]
40             else:
41                 return False
42             if getal == [] and n == a:
43                 return True
44     else:
45         if getal[-1] == '0':
46             getal = getal[:-1]
47             a = len(getal) // 2
48             n = 0
49             for z in range(0, len(getal) // 2):
50                 if getal[0] == getal[-1]:
51                     n += 1
52                     getal.reverse()
53                     getal = getal[:-1]
54                     getal.reverse()
55                     getal = getal[:-1]
56                 else:
57                     return False
58                 if getal == [] and n == a:
59                     return True
60         else:
61             return False
```

```
62  def repeater(getal):
63      if getal == 8740874:
64          return True
65      getal = str(getal)
66      getal = list(getal)
67      if len(getal) // 2 == len(getal) / 2:
68          n = 0
69          while n != len(getal) // 2:
70              if getal[n] == getal[n + len(getal) // 2]:
71                  n += 1
72              else:
73                  return False
74          return True
75      else:
76          return False
77
78  def radarrepeater(getal):
79      if radar(getal) == True and repeater(getal) == True:
80          return True
81      return False
82
83  def numismatist(lijst, soort = None):
84      if soort is None:
85          soort = standvastig
86      if soort == standvastig:
87          for i in lijst:
88              if standvastig(i):
89                  i == [i]
90                  lijst += i
91      elif soort == radar:
92          for i in lijst:
93              if radar(i):
94                  i = [i]
95                  lijst += i
96      elif soort == repeater:
97          for i in lijst:
98              if repeater(i):
99                  i = [i]
100                 lijst += i
101     elif soort == radarrepeater:
102         for i in lijst:
103             if radarrepeater(i):
104                 i = [i]
105                 lijst += i
106     else:
107         for i in lijst:
108             if soort(i):
109                 i = [i]
110                 lijst += i
111     return lijst
```

(b) Analysis

```
Error at row 89, column 17
  Incompatible types for operation ==
  The following combinations exist:
  Combination 0: int == list[int, str]
    Left side has type int
    In the following cases:
      Call to numismatist at row 113, column 1
      Element of the collection at row 87, column 9
      Assignment to lijst at row 113, column 1
      Assignment to i at row 87, column 9

    Right side has type list[int, str]
    In the following cases:
      Always


  Combination 1: str == list[int, str]
    Left side has type str
    In the following cases:
      Call to numismatist at row 113, column 1
      Element of the collection at row 87, column 9
      Assignment to lijst at row 113, column 1
      Assignment to i at row 87, column 9

    Right side has type list[int, str]
    In the following cases:
      Always
```

```
Error at row 63, column 8
  Incompatible types for operation ==
  The following combinations exist:
  Combination 0: str == int
    Left side has type str
    In the following cases:
      Call to numismatist at row 113, column 1
      Element of the collection at row 97, column 9
      Call to repeater at row 98, column 16
      Assignment to lijst at row 113, column 1
      Assignment to i at row 97, column 9
      Assignment to getal at row 98, column 16

    Right side has type int
    In the following cases:
      Always
```

```
Warning at row 93, column 16
  Not all code paths have returned a value
  Python will pretend a None value was returned
  In the following cases:
  Case 1
    Condition at row 25, column 5 is false
    Condition at row 29, column 5 is true
    Condition at row 33, column 13 is true

  Case 2
    Condition at row 25, column 5 is false
    Condition at row 29, column 5 is true
    Condition at row 41, column 13 is false

  Case 3
    Condition at row 25, column 5 is false
    Condition at row 29, column 5 is false
    Condition at row 44, column 9 is true
    Condition at row 49, column 17 is true

  Case 4
    Condition at row 25, column 5 is false
    Condition at row 29, column 5 is false
    Condition at row 44, column 9 is true
    Condition at row 57, column 17 is false
```

```
Warning at row 92, column 9
  Some code paths change the collection that's being iterated over
  This can have unexpected consequences
  In the following cases:
  Case 1
    Condition at row 93, column 13 is true
    Assignment to i at row 94, column 17
    Assignment to lijst at row 95, column 17
```

Figure 5.7 is about as large of a submission as one can comfortably put on paper. It also served as a performance stress test, as the analysis that Fosite does has an exponential complexity. The submission also uses all the non-trivial aspect of the interpreter, such as default arguments, function objects, list concatenation, and loops. Evaluating this submission takes about 60 ms on an 2.2 GHz i5-5200U. Only 20 ms of those 60 ms are spent in the analysis itself – the rest is spent in parsing and transforming the input.

Apart from the curious check on line 30, there are some interesting mistakes in the submission as well. It's possible that none of the return statements in the `radar` function ever get executed, perhaps because the check on line 42 should get swapped with the `else` on line 40. Line 89 is interesting because students write `==` instead of `=` more often than one might expect. Reusing the same identifier actually helps the student here, as line 90 would result in an type error during normal execution – which would make clear that line 89 doesn't assign to `i`. The other messages such as the loop on line 92 are even more problematic, as line 95 causes an endless loop.

Chapter 6

Conclusion and Future Work

We have developed an abstract interpreter that can successfully recognize some common, but non-trivial, mistakes that students make. The error messages describe the circumstances in which a problem may arise so that users are more likely to agree with the analysis. There are other interesting patterns the analysis currently doesn't recognize, even though they could prove useful. Unused variables for example can be an indication that the user forgot to implement some functionality. The rationale behind linters is that experience has shown that some patterns lead to errors – and experiences should be the driving force behind expanding our feature set as well.

The results have shown that students sometimes struggle with testing conditions in loops – where they use a separate `while` loop in the outer loop instead of a simple `if` statement. The current analysis warns about the potential endless loops it may cause, but it would be even better if it also explains what the current code does, proposes a different solution, and explains the difference between the two versions. This is an idea that the developer Clippy, the linter tool for Rust, proposed in his blog.[1] The goal is to have an automated system that provides feedback in the same way an educator would – by also taking into account the context in which an error occurs.

Abstract interpreters have already been used to perform an aliasing analysis to optimize dynamic languages [12]. The authors accredit their succes to three characteristics of their analysis:

- *Flow-Sensitivity*: The order of the statements are taken into account.
- *Type sensitivity*: Through the use of an abstract interpreter and their own typing system.
- *Context sensitivity*: Distinguishing between different calling contexts. They achieve this using inlining, which is equivalent to the function calls we perform.

[1] http://manishearth.github.io/blog/2017/05/19/teaching-programming-proactive-vs-reactive/

Our own analysis shares all these characteristics, most of them come with the use of an abstract interpreter, while also being inherently path-sensitive. This gives us confidence that our own data-flow analysis can be successfully applied to code transformations as well. The analysis is done in a matter of milliseconds on large submissions, which leaves a lot of room to work with. Additional analysis features might include detecting dead code or simplifying nested code branches.

Namespaces and heterogeneous collections are accurately modeled in an path-sensitive fashion, but all branch points are currently presumed to be independent. Adding symbolic execution may significantly improve the path-sensitivity. This is expected to be a challenging endeavour. Our approach only solves the heterogenous collection problem that was highlighted in code sample 12 – the other challenges are still there. On top of that, representing the abstract values in a statically typed language can be a software architectural challenge as well.

References

[1] Alessandro Bruni, C.F., Tim Disney A peer architecture for lightweight symbolic execution.

[2] Alpern, B. et al. 1988. Detecting Equality of Variables in Programs. *Conference Record of the 15th Annual Symposium on Principles of Programming Languages* (1988), 1–11.

[3] Ball, T. and Daniel, J. 2015. *Deconstructing dynamic symbolic execution*. IOS Press.

[4] Bessey, A. et al. 2010. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*. 53, 2 (Feb. 2010), 66–75.

[5] Cadar, C. et al. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. *Proceedings of the 8th usenix conference on operating systems design and implementation* (Berkeley, CA, USA, 2008), 209–224.

[6] Carey and Sturm *Space software*.

[7] Cooper, K.D. et al. 2001. A simple, fast dominance algorithm. *Software Practice & Experience*. (2001).

[8] Copeland, T. 2005. *PMD applied: An easy-to-use guide for developers*. Centennial Books.

[9] Crockford, D. 2008. *JavaScript: The good parts*. O'Reilly Media, Inc.

[10] Cytron, R. et al. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (Oct. 1991), 451–490.

[11] Fowler, M. 1999. *Refactoring: Improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc.

[12] Gorbovitski, M. et al. 2010. Alias analysis for optimization of dynamic languages. *Proceedings of the 6th symposium on dynamic languages* (New York, NY, USA, 2010), 27–42.

[13] Gregg, B. Node.js in flames.

[14] Hand, J.A. 1971. *Guidance, navigation and control*. MIT.

[15] ISO/IEC 2003. Rationale for international standard– programming languages– c.

[16] Jensen, S.H. et al. 2009. Type analysis for javascript. *Proceedings of the 16th international symposium on static analysis* (Berlin, Heidelberg, 2009), 238–255.

[17] Jukka Mypy documentation.

[18] Julien Verlaguet, A.M. Hack: A new programming language for hhvm.

[19] Kamiya, T. et al. 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.* 28, 7 (Jul. 2002), 654–670.

[20] Kotzmann, T. et al. 2008. Design of the java hotspot&Trade; client compiler for java 6. *ACM Trans. Archit. Code Optim.* 5, 1 (May 2008), 7:1–7:32.

[21] Kreide, H. and Lambert, D.W. 1964. *Computation: Aerospace computers in aircraft, missiles, and spacecraft.*

[22] Livshits, B. et al. 2015. In defense of soundiness: A manifesto. *Commun. ACM.* 58, 2 (Jan. 2015), 44–46.

[23] Lutz, M. 1996. *Programming python.* O'Reilly & Associates, Inc.

[24] Madsen, M. 2015. *Static analysis of dynamic languages.* Aarhus University.

[25] Matthias Rieger, U. of B., Stéphane Ducasse Software Composition Group Visual detection of duplicated code.

[26] Moura, L. de and Bjørner, N. 2008. Z3: An efficient smt solver. *Tools and algorithms for the construction and analysis of systems: 14th international conference, tacas 2008, held as part of the joint european conferences on theory and practice of software, etaps 2008, budapest, hungary, march 29-april 6, 2008. proceedings.* C.R. Ramakrishnan and J. Rehof, eds. Springer Berlin Heidelberg. 337–340.

[27] Novillo, D. 2007. Memory ssa-a unified approach for sparsely representing memory operations. *Proc of the gcc developers' summit* (2007).

[28] Novillo, D. 2003. Tree ssa a new optimization infrastructure for gcc. *In proceedings of the 2003 gcc developers* (2003), 181–194.

[29] Păsăreanu, C.S. and Rungta, N. 2010. Symbolic pathfinder: Symbolic execution of java bytecode. *Proceedings of the ieee/acm international conference on automated software engineering* (New York, NY, USA, 2010), 179–180.

[30], Paul Hudak et al. 2007. A history of haskell: Being lazy with class. (June 2007).

[31] Rhodes, D. et al. 2014. Dynamic detection of object capability violations through model checking. *SIGPLAN Not.* 50, 2 (Oct. 2014), 103–112.

[32] Ritchie, D.M. 1993. The development of the c language. *The second acm sigplan conference on history of programming languages* (New York, NY, USA, 1993), 201–208.

[33] Sui, Y. and Xue, J. 2016. SVF: Interprocedural static value-flow analysis in llvm. *Proceedings of the 25th international conference on compiler construction* (New York, NY, USA, 2016), 265–266.

[34] Visser, W. et al. 2003. Model checking programs. *Automated Software Engg.* 10, 2 (Apr. 2003), 203–232.

[35] Wang, X. et al. 2012. Undefined behavior: What happened to my code? *Proceedings of the asia-pacific workshop on systems* (New York, NY, USA, 2012), 9:1–9:7.

[36] Wegman, M.N. and Zadeck, F.K. 1991. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.* 13, 2 (Apr. 1991), 181–210.

[37] Yang, W. et al. 1989. *Detecting program components with equivalent behaviors*.

[38] Yichen Xie, B.H., Mayur Naik Soundness and its role in bug detection systems.

[39] Yunhui Zheng, S.S., Vijay Ganesh Z3str2: An efficient solver for strings, regular expressions, and length constraints.

[40] A guide to undefined behavior in c and c++. http://blog.regehr.org/archives/213.

[41] Alias improvements branch. https://gcc.gnu.org/wiki/Alias_Improvements.

[42] Crockford on javascript - section 8: Programming style & your brain. https://www.youtube.com/watch?v=taaEzHI9xyY.

[43] Dynamic languages make for harder to maintain large codebases. http://softwareengineering.stackexchange.com/a/221658.

[44] Epigrams on programming. http://www-pu.informatik.uni-tuebingen.de/users/klaeren/epigrams.html.

[45] Is open mpi 'valgrind-clean' or how can i identify real errors? https://www.open-mpi.org/faq/?category=debugging#valgrind_clean .

[46] JSLint. https://github.com/douglascrockford/JSLint.

[47] MemorySSA. http://llvm.org/docs/MemorySSA.html.

[48] The rpython toolchain. http://rpython.readthedocs.io/en/latest/translation.html#flow-graphs.

[49] What every c programmer should know about undefined behavior. http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html.