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].

```python
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

```python
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

```python
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)*,* $p \prec q \iff n_p \prec_{lex} n_q \lor (n_p = n_q \land b_p \prec b_q)$*.*

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* $\{\, (n_p, i, t_p) \mid 0 \leq i < t \land i \neq b_p \,\}$*. 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.