Static Program Analysis of Data Usage Properties

Master's Thesis Project Description

Simon Wehrli
supervised by Dr. Caterina Urban

March 7, 2017

1 Introduction

In the past years, the Python programming language and associated libraries became popular for data science applications due to the simple syntax and broad integration with common data science libraries. For quick data inspection, the dynamic nature of Python seems very attractive but introduces some challenges for analysis tools. Current tools mainly aim at detecting programming errors that lead to unhandled runtime exceptions. But programming errors that do not trigger exceptions at all can still have serious consequences, since the erroneous treatment or the omitting of data items can lead to false results while showing no indication of the faulty derivation of those results.

Our goal is to analyse data usage properties with a static analysis of the program, so without presuming concrete input data that exposes a faulty behaviour. We target a sound approach that gives us mathematical guarantees of the specified properties about any syntactically correct program and input data.

Consider the following Python program, which calculates the average over some body height data read from a tab-separated file, annotated with the gender. For one input line the gender information is missing and due to a bug, this data item is not included in the calculation.

```
from functools import reduce
import fileinput

def average(data):
    values = data['m'] + data['f']
    return reduce(lambda x, y: x + y, values) / len(values)

if __name__ == "__main__":
    data = {}

    for line in fileinput.input():
        (name, gender, body_height) = line.strip().split('\t')
        if not gender in data:
            data[gender] = []
        data[gender].append(int(body_height))

    print(data)  # {'f': [153, 169, 166], 'm': [188, 169], '': [178]}
    print(average(data))  # 169.0
```

```
Julia    f       153
Beat     m       188
Manuel   m       169
Sabine   f       169
Philip           178
Laura    f       166
```

To detect such problems we have to analyse the *potential effect* of all input data items and report unexpected cases, such as no usage of some data items.

2 Assumptions

We assume that the programs

- are deterministic,
- are single-threaded,
- terminate on any correctly formatted input,
- have a single pair of an initial and a final control point (before the initial and after the final statement, respectively),
- come type annotated[1].

3 Core Goals

We list the mandatory tasks (on the right side, a rough estimate for the time that we allocate to the respective tasks):

- Define what is considered as an input data item to the analysed program. The simplest approach, without the need for user input, is to assume that every line of input read from Python's standard input `sys.stdin` is a data item. Or, as in the example above, we could assume that the input has a tabular form and each tab is a data item. This allows evaluating the analysis on programs with a more structured input (e.g. relational). $(\star\star)$
  The size of the input is assumed to be finite but its size is not statically known. This adds a real challenge compared to related analysis, such as the truly-live variable analysis (see Related Work), which assume that the number of items of interest is statically known (e.g. all variables in the program).

[1]Type annotations are available in Python ≥ 3.5, `https://docs.python.org/3.5/library/typing.html`

- Formalize the intuition of the property "An input data item is used/unused". ($\star\star$)
  A definition for a fixed number of input data items is given here. Let $\mathcal{X}$ be the finite set of program variables. Each variable $X_i \in \mathcal{X}$ has a (possibly infinite) domain D_i of valid values with an equivalence relation $=$. To ease explanation we assume that the fixed number of inputs and outputs are stored in specific variables $\mathcal{V} \subseteq \mathcal{X}$ and $\mathcal{U} \subseteq \mathcal{X}$, respectively.
  Let Σ be the set of all program states. A program *state* $s \in \Sigma$ is a pair consisting of a program label and an environment that defines the values of the variables $\mathcal{X}$ at the program control point designated by the label. We use the shorthand notation $s(X_i)$ to refer to the value of X_i in the environment belonging to state s. Let $\mathcal{I} \subseteq \Sigma$ be the set of *initial program states*, which are the states with a label designating the initial control point *before* the first statement of the program. Analogously, $\Omega \subseteq \Sigma$ is the set of *final program states* with a label designating the final control point *after* the final statement. The environments of initial states must be valid in the operational semantics of the program. A state t is called *reachable* from state s, denoted by the predicate $\mathit{reach}(s,t)$, if there is a trace from s to t.
  The variable V_i is considered *used* by a program if and only if there exist two initial program states $s, s' \in \mathcal{I}$ with equal values for all input variables except the input value for V_i and there are two final program states $t, t' \in \Omega$, *reachable* from s, s', respectively, with unequal values for at least one of the output variables. Formally,

$$\begin{aligned} \mathit{used}(V_i) \equiv\ & \exists s, s' \in \mathcal{I}, t, t' \in \Omega : \mathit{reach}(s,t) \wedge \mathit{reach}(s',t') \\ & \wedge \forall V_j \in \mathcal{V} : (s(V_j) = s'(V_j) \leftrightarrow j \neq i) \wedge \exists U_k \in \mathcal{U} : t(U_k) \neq t'(U_k). \end{aligned}$$

  Similar properties have been stated and built analysis for in the context of secure information flow [4].

- Design a static analysis within the framework of Abstract Interpretation [8] that approximates the solution to the question: Does a data item fulfill the *used* property defined above? In particular, this requires the design of an abstract domain, transfer functions for the statements/expressions of Python programs (or a subset of it) and also the finding of a suitable widening operator. ($\star\star\star$)
  It must be figured out how far the *Interprocedural Dataflow Analysis* approach [9] can be adapted. In the cited paper, a certain class of problems is solved, that includes the problem of finding truly-live variables, which is closely related to our problem.

- Implement the designed analysis (in Python) and evaluate the analysis on a bunch of example programs created manually and deducted, for instance, from the *Data Programming* course at the University of Washington [1]. ($\star\star$)

4 Extensions

We list the following tasks as possible extensions (on the right side, a rough estimate for the time that we allocate to the respective tasks):

- Extend the class of analyzable programs. So far we required the programs to have an certain structure of input. We could extend that to either let the user specify the structure in some simple specification language or we could try to derive the structure from the program itself. (⋆⋆⋆)
- Support external data science libraries (the NumFOCUS nonprofit organisation lists some of them[2]). A possibility is to use appropriate annotations to library function calls. (⋆⋆)
- Extend the analysis to more sophisticated properties. While the used/unused property helps finding bugs where data items have no effect on the result, we could be interested in a notion of "strength of effect" of data items. For instance, this could allow us to formulate and check data usage properties such as "every data point is weighted equally". (⋆⋆⋆)
- Use testing on concrete inputs to reduce the number of false positive alarms. Whenever our analysis reports an unused data item, we can potentially qualify this as a false alarm by testing with (cleverly choosen) concrete input values that show that the result indeed depends on the data item in question. (⋆⋆)
- Rigorously prove the existence of a hierarchy of Galois connections between the real program semantics and the choosen program abstraction following the calculational approach of Abstract Interpretation [7]. (⋆⋆⋆⋆)

5 Related Work

As mentioned already, the *Interprocedural Dataflow Analysis* [9] solves, among others, the truly-live variable problem. In short, this is the problem of determining if a variable is *transitively* live, that is, if the value written to a variable can flow to defined output statements. The approach in [9] uses a graph reachability algorithm. By encoding the required property in a graph, they heavily rely on the fact that the number of variables is known statically, which is not the case for our problem.

A work with a similar goal is CheckCell [3] which analyses the data usage on Microsoft Excel spreadsheets. While they also try to find anomalies in the effect of data input (cells), their approach is different. CheckCell uses a probabilistic approach to estimate the impact of input cells on result cells by looking at the changes occurring in the result cells when changing the input cells. It does not find errors in the spreadsheet if the data does not expose them. Instead, we will focus on the program rather than the data and prove data usage properties independently from concrete data.

Another work [6] builds a static analysis for Excel spreadsheets that detects type-unsafe operations that are possible in Excel due to its weak type system. This focuses on the program, but only on finding errors introduced by unintended run-time type-unsafe operations. Since we assume that our programs come type annotated, we will assume those errors are already ruled out. So the problems solved in the paper and our stated problem are orthogonal.

An only partial related paper [5] discusses a static analysis to determine if a program represents a continous function, or equivalently, if infinitesimal changes to its inputs can only cause infinitesimal changes to its outputs. This is related in the sense that we also want to inspect the effect of the inputs on the outputs, except that we are satisfied with *any* potential effect on the output. In addition, their analysis is targeted at very restricted pseude code.

References

[1] Cse160: Data programming. `http://courses.cs.washington.edu/courses/cse160/`, 2017. [Online; accessed 27. February 2017].

[2] NumFOCUS. `http://www.numfocus.org/`, 2017. [Online; accessed 27. February 2017].

[3] Daniel W Barowy, Dimitar Gochev, and Emery D Berger. Checkcell: data debugging for spreadsheets. In *ACM SIGPLAN Notices*, volume 49, pages 507–523. ACM, 2014.

[4] Gilles Barthe, Pedro R D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE*, pages 100–114. IEEE, 2004.

[5] Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. Continuity analysis of programs. In *ACM Sigplan Notices*, volume 45, pages 57–70. ACM, 2010.

[6] Tie Cheng and Xavier Rival. Static analysis of spreadsheet applications for type-unsafe operations detection. In *European Symposium on Programming Languages and Systems*, pages 26–52. Springer, 2015.

[7] Patrick Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science*, 277(1-2):47–103, 2002.

[8] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[9] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61. ACM, 1995.