Automatic Verification of Closures and Lambda-Functions in Python

Master's Thesis Project Description

Benjamin Weber

March 2017

Introduction

Higher-order functions take other functions as parameters. Thus, higher-order functions can leave parts of their functionality open for later customization or extension. Examples from the functional programming paradigm are **map**, **filter** and **fold**. Being able to customize the functionality of those utilities yields powerful tools for working with lists. However, they require that at least their parameter variables can be bound to functions. The possibility to bind variables to functions introduces a lot of flexibility to a programming language, but also complicates program verification. Given that a variable could be bound to any function, the precondition and postcondition of that function may only be known at runtime.

Closures are functions (named or anonymous) that may have captured variables. They increase flexibility since functions are given access to variables that may be relevant to their functionality. From a verification perspective, closures complicate reasoning because they can capture local variables. It is then unclear how to refer to those captured local variables in a closure's contracts. In particular, when functions can mutate those captured variables, it is often essential that their contracts can refer to those variables.

Viper and Nagini

Viper [6] is an automatic, deductive verification infrastructure developed at the Chair of Programming Methodology at ETH Zürich. Verification is done through its intermediate verification language: the Viper language. There are several front ends for Viper which translate programs written in programming languages and annotated with verification constructs to the Viper language. Nagini is a front end dedicated to Python programs with static type annotations.

This Master's thesis aims to extend Nagini with the necessary means to allow for formal verification of lambda functions and closures.

Motivating Examples

```
1 def func(g, a, b, c):
2     Requires(True)
3     Ensures(True)
4     while a.parameter < b + c:
5         a.parameter = a.parameter * a.parameter
6     # g reads a.parameter and writes to a.result
7     e, f = g(a, b, c)
8     return 1/a.result, sqrt(e), log(f)
```

(a) Specification usage.

```
def f(setup, compute):
  setup()
  compute()
```

(b) Unknown specifications.

```
def factory():
  x = 0

  def increment():
    nonlocal x
    x += 1
    return x

  return increment
```

(c) Hidden variables.

Figure 1: Examples demonstrating challenges of closure verification.

Given the examples in Figure 1, we would like to verify that those methods adhere to some provided specifications. Additionally, we want to verify other correctness properties such as memory safety or concurrency safety.

In example 1a, at line 7 we can only establish that $\texttt{a.parameter} \geq \texttt{b} + \texttt{c}$. Therefore, `g`'s precondition must be implied solely by $\texttt{a.parameter} \geq \texttt{b} + \texttt{c}$. In order to verify line 8, it must hold that $\texttt{a.result} \neq 0 \wedge \texttt{e} \geq 0 \wedge \texttt{f} > 0$. Thus, `g`'s postcondition has to imply that expression. These restrictions on `g` should be expressible in `func`'s precondition.

One way to verify example 1b, is to specify a precondition for method `f` that implies the precondition of `setup`. The postcondition of `setup` could be required to imply `compute`'s precondition. Finally, we could specify a postcondition for method `f` that is implied by `compute`'s postcondition. To increase flexibility, the contracts of `f` should be generic w.r.t. `setup` and `compute` while still allowing to impose the mentioned restrictions on `setup` and `compute`.

In Figure 1c, we need to be able to specify `increment` which gets returned from `factory` as a closure. In particular, `factory`'s specification has to be able to specify the returned closure. In this case, the semantics of `increment` depend on the captured local variable `x` which is only visible inside `factory`'s scope. Because `increment` is returned and thus made accessible outside of `factory`'s scope, it is unclear how to specify `increment` without referring to the hidden variable `x`.

Core Goals

1. Explore the state of the art regarding verification of closures

The first core goal consists of various steps to increase the understanding of verification of closures in the Viper setting. The state of the art should be reviewed to get an idea of what has already been done in this direction of research and to find out what worked well in particular. This includes research papers (e.g., [3, 5] among others) as well as concrete implementations (e.g., Why3 [1], VeriFast [2], Chalice [4], Viper [6] and others). Motivating examples should be gathered that highlight aspects of closures which need to be taken into consideration for verification thereof. Each motivating example should highlight exactly one of such aspects.

2. Develop a methodology for specification and verification of Python functions (as first-class citizens)

In this core goal, we want to create a methodology to specify and verify Python functions that can be bound to variables, which means that their contracts are no longer known statically. Figure 1 highlights some of the challenges that we want to overcome. Supporting variable capturing is excluded from this core goal (see extension goal 1).

As a first step, the methodology should focus only on side-effect free functions. In a second step, support for heap modifying functions should be added.

3. Encoding and verifying closure specifications in Viper

In parallel to developing a methodology for verification and specification of Python closures (core goal 2), an encoding of that methodology to the Viper language has to be created. That is, the constructs used in that methodology have to be expressed in the Viper language. The encoding should strive to allow for automation during verification.

4. Design the necessary syntactic elements for reasoning about closures for Nagini

Viper and Nagini aim to be practical and user friendly. These properties are affected by the syntax of the annotations used for verification. In particular, the syntax should allow for a concise and readable expression of program semantics. It should fit well into the existing syntax and one should be able to combine it with the already established features of Nagini.

5. Implement the translation from the Python code with specifications to the Viper language within Nagini

The previous core goals naturally lead to this last core goal: The actual implementation within Nagini. The implementation should work well with the existing code. Furthermore, error messages from Viper should be translated back into the context of the Python code.

Extension Goals

1. Enable verification and specification of closures with capturing

Example 1c demonstrates how closures that capture local variables complicate verification. In this extension goal, the methodology from core goal 2 should be extended to support closures that capture variables. The extension should then also be implemented as part of Nagini.

2. Verify SCION's use of closures and lambda functions

SCION [9] is a clean-slate approach to internet architecture addressing many issues of today's internet. In a joint effort the Network Security Group, the Information Security Group and the Chair of Programming Methodology will try to verify SCION's properties from the high-level design all the way down to its implementation. Since its implementation is written in Python, Nagini will play a crucial role in the verification of that implementation. One of the motivations for this Master's thesis is, in fact, the use of closures and lambda functions in its codebase. As such, a suitable extension goal would be to verify the components that make use of closures and lambda functions.

3. Combine verification of closures with other advanced features

Closure verification and specification may or may not be compatible with other advanced verification features such as quantified permissions [7] or magic wands [8].

In this extension goal, we want to explore whether closure verification and specification is compatible with other advanced features, how they could be made compatible and whether it would be beneficial to have closure verification and specification in combination with other advanced features.

References

[1] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In *European Symposium on Programming*, pages 125–128. Springer, 2013.

[2] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods Symposium*, pages 41–55. Springer, 2011.

[3] Ioannis T Kassios and Peter Müller. Specification and Verification of Closures. Technical report, ETH Zürich, 2010. `http://pm.inf.ethz.ch/publications/getpdf.php?bibname=Own&id=KassiosMueller10.pdf`.

[4] K. R. M. Leino, P. Müller, and J. Smans. Verification of Concurrent Programs with Chalice. In A. Aldini, G. Barthe, and R. Gorrieri, editors, *Foundations of Security Analysis and Design V*, volume 5705 of *Lecture Notes in Computer Science*, pages 195–222. Springer-Verlag, 2009.

[5] F. Meier. Closure verification in an automated fractional permission setting. Bachelor's thesis, ETH Zürich, 2014. `https://www.ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Fabian_Meier_BA_report.pdf`.

[6] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

[7] Peter Müller, Malte Schwerhoff, and Alexander J Summers. Automatic verification of iterated separating conjunctions using symbolic execution. In *International Conference on Computer Aided Verification*, pages 405–425. Springer, 2016.

[8] Malte Schwerhoff and Alexander J Summers. *Lightweight support for magic wands in an automatic verifier*, volume 37. Schloss Dagstuhl Leibniz Center for Informatics, 2015.

[9] Xin Zhang, Hsu-Chun Hsiao, Geoffrey Hasker, Haowen Chan, Adrian Perrig, and David G. Andersen. SCION: Scalability, Control, and Isolation on Next-Generation Networks. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, May 2011.