Scala.js integration with Leon to improve counterexample presentation in the web interface

Liu Fengyun
Ólafur Páll Geirsson

Abstract

Although a main feature of Leon is to provide counterexamples to invalid programs, it can remain difficult to reason about why a counterexample breaks a program. In this project, we seek to answer the following question: given a Leon program and a counterexample input, can we present the error in the code in such a way that makes it easy for the programmer to reason about why the program fails. To aid us in answering this question, we propose to integrate Scala.js with Leon and use heavily the substitution model of evaluation. Moreover, we aim to borrow ideas developed in Bret Victor's excellent essay on "Learnable Programming" [1].

1 Introduction

Currently the Leon web interface shows counterexamples in a dialog, or in a tip when user hovers the pointer over a variable. This can be unfriendly, as the programmer has to remember the variable values and do substitution and execution in mind in order to reason where the problem lies.

Why not just annotate(or substitute) the variables in the buggy function with the counterexample as input values? This way, user can easily figure out which line goes wrong by following the execution in an interactive and visual way.

The detailed solution is as follows:

- When user clicks *debug* button in the counterexample dialog, a debug box appears, with all the formal parameters in the function being annotated(or substituted) by the counterexample values.
- *Function calls* are annotated by their return values. If user clicks a function call in a debug box, another debug box will overlay current debug box. User can close a debug box by clicking [x] in the up-right corner of the debug box.

[1] `http://worrydream.com/#!/LearnableProgramming`

- *Local variables* are annotated by the values they hold. The value of *this* will be shown as environment in each debug box. *Free variables*(closure variables) in a function are correctly handled and annotated as well.

Advanced features we consider(but may beyond current scope of the project):

- User can modify the annotated program in debug box and check if current counterexample passes or not.
- User can try different values of function formal parameters in debug box, and check if the postcondition holds or not.

2 Technical Details

This project has three technical aspects:

- Implement the user interface using a front-end reactive library, such as *scalajs-react*[2]
- Parse Leon programs into an AST in order to perform correct annotation (or substitution) of Leon programs.
- Interpret the execution trace of a Leon functions given an input counterexample

We believe that the front-end programming is a manageable task, given that the user interface is not complicated. Likewise, the web interface seems to already parse Leon programs to some extent which may be useful

We are not entirely sure how to best interpret the execution trace of Leon functions given an input. We would appreciate some feedback on the feasibility of this task.

[2] `https://github.com/japgolly/scalajs-react`