Shiranui: A Live Programming with Support for Unit Testing

Tomoki Imai
Tokyo Institute of Technology, Japan
imai.t.af@m.titech.ac.jp

Hidehiko Masuhara
Tokyo Institute of Technology, Japan
masuhara@acm.org

Tomoyuki Aotani
Tokyo Institute of Technology, Japan
aotani@is.titech.ac.jp

Abstract

Live programming environments help the programmers to try out expressions by giving immediate feedback on the results as well as intermediate evaluation processes. However, the feedback is transient, and its correctness is merely confirmed by the programmers' manual inspection. We seamlessly integrate live programming with unit testing by proposing novel features (1) that converts a lively-tested expression into a unit test case, and (2) that extracts a unit test case from an execution trace of a lively-tested expression. In this poster, we overview Shiranui, our live programming environment, and present the proposed features implemented in Shiranui.

Categories and Subject Descriptors D.2.6 [*Software Engineering*]: Programming Environments—Interactive Environments

General Terms Languages, Human Factors

Keywords Live Programming, Testing, Debugging

1. Introduction

When the programmers develop a program, they constantly test functions in order to reason about their behaviors. They do so through an interactive environment or a testing framework.

A live programming environment [1] is such an interactive environment that provides feedback for every few keystrokes in the programming editor. It is useful when the programmers write functions by trial-and-error.

Testing frameworks like JUnit provide a persistent way to test function's bahaviors. It however takes time to formulate test cases for a function with proper sets of parameters and expected results, especially when functions involve with first-class function values or complicated data structures.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.

Copyright is held by the owner/author(s).

SPLASH Companion'15, October 25–30, 2015, Pittsburgh, PA, USA
ACM. 978-1-4503-3722-9/15/10
http://dx.doi.org/10.1145/2814189.2817268

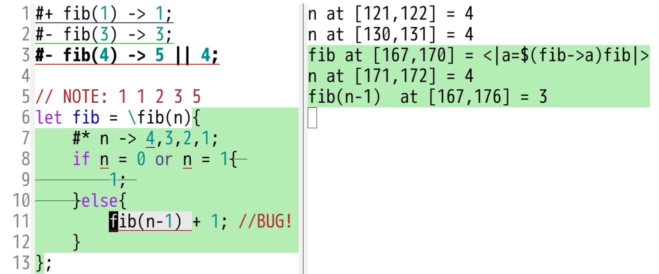

Figure 1. Screenshot of Shiranui

We propose a set of features for a live programming environment that combine advantages of both approaches: quick feedback in live programming, and persistency in unit testing. With those features, the programmers can check function's behaviors in a lively manner, and can convert the results of the checking into persistent test cases.

2. Test Supporting Features in Live Programming Environment Shiranui

We are developing Shiranui, a live programming environment, and a set of features that supports testing on top of Shiranui. Below, we first overview the basic features of Shiranui, and then discuss the test supporting features.

2.1 Overview of Shiranui

Shiranui is a live programming environment similar to YinYang [3] and Apple Swift [4]. It automatically re-executes the program while being edited, and immediately displays the program's results (including intermediate ones) as an overlay on top of the program text.

Figure 1 is a screenshot of Shiranui, consisting of a program text (left) and an environment view (right). The language is also called Shiranui, which is a dynamically-typed functional language with explicit mutable data structures. A program consists of *flylines* start with a # symbol, and top-level variable/function definitions. A flyline, whose details are explained in the later section, specifies an experimental expression to be executed[1]. Whenever the programmer edits the text, the environment automatically re-executes the ex-

[1] Since a flyline allows only one expression there, it currently does not support tests about side effects.

pressions in the flylines, and inserts the results at the end of flylines.

When the programmer issues the focus command on a flyline (say the one at line 3), Shiranui highlights with green color the fragments of code that are executed by the expression on the flyline, and also shows a value-history of variables during the execution (e.g., the numbers at line 7). The programmer can then issue the focus command on one of the values so as to examine the execution that yielded the focused value (as shown by overstrikes on unexecuted code fragments at lines 8–10 and the values of the expressions in the environment view).

2.2 Testing Features in Shiranui

Unlike other live programming environments [2], Shiranui supports the following features for connecting live programming to unit testing, namely, converting experimental expressions to test cases, converting (even complicated) values into expressions, and executing each flyline in an isolated execution environment.

With those features, the programmer can develop a program by first writing functions with experimental expressions to understand the behavior of the functions in a trial-and-error fashion, and then convert the experimental expressions into unit test cases just by issuing a command on each experimental expression. The programmer can reuse the values shown in the value-history of variables as parameters to those experimental expressions. This is particularly useful when a function involves with a complicated data structure or a function value with lexical variables. The isolated execution environment is crucial to evaluate those experimental expressions and test cases without interference.

2.2.1 Converting Experimental Expressions to Test Cases

Shiranui provides a command to quickly convert experimental expressions (what we call *idle flylines*) to unit test cases (what we call *test flylines*). It is done by just changing `#+` in a idle flyline to `#-` in a test flyline. We can also use a part of output of idle flylines to make expected results in test flylines.

When the programmer is defining a function, he or she first writes idle flylines to check the behavior of the function being written, and once they show expected results, he or she can promote them to test flylines by issuing the command.

2.2.2 Converting Complicated Values into Expressions

Shiranui can display complicated data as a valid expression, even if the value has a circular structure or contains a function value with lexical variables. This feature is useful when creating test cases with complicated data because the programmer can simply copy-and-paste once it appears in the environment view. It also enables the programmer to test even anonymous functions with free variables.

2.2.3 Executing Each Flyline with an Isolated Execution Environment

When Shiranui executes flylines, the execution environment of each expression is separated from each other. This is an important feature for testing, where test fixtures must be initialized for a run of each test case. By isolating execution environments, each test expression can assume that top-level variables initially have the same values, and can be allowed to destruct those values without affecting the executions of other expressions. Other live programming environments, Swift for example, simply execute top-level expressions in turn, which is sometimes confusing when evaluation causes side-effects on top-level variables.

3. Conclusion and Future Work

We proposed a set of features that support unit testing in a live programming environment. The features include an editor command that converts experimental expressions into unit test cases, an editor command that converts complicated values into expressions, and execution of experimental and testing expressions under isolated environments. We believe that those features allow the programmer to construct functions in a “lively” fashion, at the same time to obtain persistent unit test cases with minimal efforts.

The implementation of Shiranui consists of a backend written in C++ (about 7,200 LoC) and an Emacs-based frontend written in Emacs Lisp (about 900 LoC). The backend is implemented as an interpreter, and is supposed to evaluate a whole program text for each time a character is edited in the frontend. The implementation is publicly available at `https://github.com/tomoki/Shiranui`.

Our future work includes improvement of performance and scalability, improvement of user-interface, and empirical evaluations. As for the performance and scalability, the runtime would need to re-execute only a part of a program that is affected by a change. The language would also need modularity mechanisms for describing larger applications.

References

[1] C. M. Hancock. *Real-time Programming and The Big Ideas of Computational Literacy.* PhD thesis, Massachusetts Institute of Technology, 2003.

[2] T. Imai, H. Masuhara, and T. Aotani. Shiranui: Test-friendly Live Programming Environment. In *The 31st JSSST Annual Conference*, Sep. 2014.

[3] S. McDirmid. Usable Live Programming. *Proceedings of Onward! ’13*, pages 53–62, 2013.

[4] Apple Inc. Swift - Overview - Apple Developer. `https://developer.apple.com/swift/`. Accessed 2015-6-30.

[2] It is announced that Apple Swift 2’s Playgrounds can “Create new tests and verify they work before promoting into your test suite [4].” We independently proposed our testing feature in Shiranui [2].