Making Live Programming Practical by Bridging the Gap between Trial-and-Error Development and Unit Testing

Tomoki Imai, Hidehiko Masuhara, Tomoyuki Aotani

Tokyo Institute of Technology, Japan
Background: Introduction to Live Programming

Q. What is a live programming environment?
Background: Introduction to Live Programming

Q. What is a live programming environment?

A. A programming environment, which provides immediate feedback on source code changes.
Background: Introduction to Live Programming

Q. What is a live programming environment?

A. A programming environment, which provides immediate feedback on source code changes.

Traditional Programming
All transitions are manual.

![Diagram showing the traditional programming process with arrows pointing from "Edit" to "Compile", and then to "Run", and finally to "Check".]
Background: Introduction to Live Programming

Q. What is a live programming environment?

A. A programming environment, which provides immediate feedback on source code changes.

Traditional Programming
All transitions are manual.

Live Programming
- Edit => Live System
- Live System => Check

are automatically done.
Background: Two Styles of Live Programming

We focus on “1. Re-evaluate style.”

1. Re-evaluate
Environments re-evaluate the program when source code or data changes.
- YinYang [McDirmid, 2013]
- Apple Swift’s Playground
- Our Prototype Shiranui

2. Fix-and-continue
We can change the source code of running programs. (by hot-swapping)
- Smalltalk
- Scratch
- Sonic Pi
Our Motivation: Make Live Programming Practical

Our Motivation
We want to use live programming environments in practical programming, which require:
- many functions and experiments,
- ensuring that the program works correctly.

Currently,
Live programming environments are mainly used for:
- running samples,
- small programs,
- checking library functions’ behavior.
Question:

Q. Are there any problems when we use existing re-evaluate style live programming environments in practical programming?
Q. Are there any problems when we use existing re-evaluate style live programming environments in practical programming?

A. Yes, there are (at least) three problems:
   1. single entry point,
   2. no support for testing frameworks, and
   3. no support for making small sub problems.
Problem 1: Single Entry Point

Existing live programming environments have only one entry point.

1. long feedback loop, e.g., we cannot get \( \text{sum}_\text{ave}(3) \)'s feedback before \( \text{sum}_\text{ave}(10000) \).

2. combined runtime log, e.g., \( \text{sum}_\text{ave}(10000) \)'s log and \( \text{sum}_\text{ave}(3) \)'s one are merged.

3. lost feedback. e.g., \( \text{sum}_\text{ave}(0) \) causes error, and \( \text{sum}_\text{ave}(50) \)'s feedback is lost.

=> Not suitable for large programs.
Problem 1: Single Entry Point

Existing live programming environments have only one entry point.

It’s like a big “main” function.

```scala
void main() {
  func sum_ave(n: Int) -> Int {
    var r = 0
    for i in (0 ... n) {
      r += i
    }
    return r / n
  }

  sum_ave(10000) // takes time
  sum_ave(3)
  sum_ave(0)   // Execution was interrupted,...
  sum_ave(50)
}
```
Problem 1: Single Entry Point

Existing live programming environments have only one entry point.

It’s like a big “main” function.

```plaintext
void main() {
    func sum_ave(n: Int) -> Int {
        var r = 0
        for i in (0 ... n) {
            r += i
        }
        return r / n
    }
    sum_ave(10000) // takes time 5000
    sum_ave(3) 2
    sum_ave(0) Execution was interrupted... error
    sum_ave(50)
}

=> Not suitable for large programs.
```

It causes:

1. long feedback loop,
2. combined runtime log,
3. lost feedback.

Not suitable for large programs.
Problem 1: Single Entry Point

Existing live programming environments have only one entry point. It’s like a big “main” function.

```swift
void main() {
    func sum_ave(n: Int) -> Int {
        var r = 0
        for i in (0 ... n) {
            r += i
        }
        return r / n
    }

    sum_ave(10000) // takes time
    sum_ave(3)
    sum_ave(0) // Execution was interrupted...
    sum_ave(50)
}
```

It causes:
1. long feedback loop,
   - ex. We cannot get `sum_ave(3)`’s feedback before `sum_ave(10000)`.
2. combined runtime log,
3. lost feedback.

=> Not suitable for large programs.
Problem 1: Single Entry Point

Existing live programming environments have only one entry point. It’s like a big “main” function. It causes:

1. long feedback loop,
   - ex. We cannot get `sum_ave(3)`’s feedback before `sum_ave(10000)`.

2. combined runtime log,
   - ex. `sum_ave(10000)`’s log and `sum_ave(3)`’s one are merged.

3. lost feedback.

```swift
void main() {
    func sum_ave(n: Int) -> Int {
        var r = 0
        for i in (0 ... n) {
            r += i
        }
        return r / n
    }

    sum_ave(10000) // takes time
    sum_ave(3)
    sum_ave(0) // Execution was interrupted... error
    sum_ave(50)
}
```

=> Not suitable for large programs.
Problem 1: Single Entry Point

Existing live programming environments have only one entry point.

It’s like a big “main” function.

```c
void main() {
    func sum_ave(n: Int) -> Int {
        var r = 0
        for i in (0 ... n) {
            r += i
        }
        return r / n
    }

    sum_ave(10000) // takes time
    sum_ave(3)
    sum_ave(0)  // Execution was interrupted...
    sum_ave(50)
}
```

=> Not suitable for large programs.

It causes:

1. Long feedback loop,
   - ex. We cannot get `sum_ave(3)`’s feedback before `sum_ave(10000)`.

2. Combined runtime log,
   - ex. `sum_ave(10000)`’s log and `sum_ave(3)`’s one are merged.

3. Lost feedback,
   - ex. `sum_ave(0)` causes error, and `sum_ave(50)`’s feedback is lost.
Problem 2: No Support for Testing Frameworks

“Tests” in live programming environments are transient.

- We need to check all return values by ourselves when we change source code, because it might change existing functions’ behavior.

If we add testing frameworks like JUnit, “liveness” is lost.

- Constructing expected values is not “live way.”
- Promoting experiments to test cases is not supported.
Problem 3: No Support to Make Small Sub Problem

Generally speaking, (not only live programming)

When debugging large programs, we must create small programs which reproduce the bugs.

- It is not easy especially when programs contain first-class function.

```
func fact(n: Int, cont: Int -> Int) -> Int {
    if n == 0 {
        return cont(1) // ← What occurred?
    } else {
        return fact(n-1, cont:{r in cont(r + n)})
    }
}

print(fact(10, cont: {r in r}))
```

56
Our Solution and Design

We show our prototype named **Shiranui**.

Problems and Solutions

1. Single entry point  
   => Isolated execution point
2. No support for unit testing  
   => Integrated unit testing features
3. No support for making small sub problems  
   => Shortcut to take function call out from runtime log
Solution 1: Isolated Execution Point
Shiranui executes some parts of programs in isolated interpreters parallelly. It enables:

- faster feedback,
- simpler execution logs,
- not propagating errors

```
1  let sum_ave = \(n)\{ 
2      let r = ref 0;
3      for i in [1..n]{
4          !r -> 0,1,3,6,10;
5          r <- !r + i;
6      }
7      return !r / n;
8  };
```

1. Virtually duplicate programs for each isolated execution point (L:1, 2, 3, 4).
2. Run programs parallelly and record logs separately.
3. Give feedback to users as threads are finished.
Solution 1: Isolated Execution Point
Shiranui executes some parts of programs in isolated interpreters parallelly. It enables:

- faster feedback,
- simpler execution logs,
- not propagating errors

1. Virtually duplicate programs for each isolated execution point (L:1,2,3,4).

```plaintext
let sum_ave = \(n\)=> {
    let r = ref 0;
    for i in [1..n]{
        r <= !r + i;
    }
    return !r / n;
}
```

```plaintext
let sum_ave = \(n\){
    // copied body
    sum_ave(10000);
}
```

```plaintext
let sum_ave = \(n\){
    // copied body
    sum_ave(3);
}
```
Solution 1: Isolated Execution Point
Shiranui executes some parts of programs in isolated interpreters parallelly. It enables:

- faster feedback,
- simpler execution logs,
- not propagating errors

1. Virtually duplicate programs for each isolated execution point (L:1,2,3,4).
2. Run programs parallelly and record logs separately.

```javascript
let sum_ave = \(n\) {
  let r = ref 0;
  for i in [1..n] {
    \*r -> 0,1,3,6,10;
    r <- !r + i;
  }
  return !r / n;
}
```
Solution 1: Isolated Execution Point

Shiranui executes some parts of programs in isolated interpreters parallelly. It enables:

- faster feedback,
- simpler execution logs,
- not propagating errors

1. Virtually duplicate programs for each isolated execution point (L:1,2,3,4).
2. Run programs parallelly and record logs separately.
3. Give feedback to users as threads are finished.
Solution 2: Integrated Unit Testing Features

Unit testcases are expressed as isolated execution points.

Normal execution point
- ex. twice(1) returns 1.

Successful testcase
- ex. twice(2) should return 4 and actually returns 4.

Failed testcase
- ex. twice(3) should return 6 but actually returns 9.

```javascript
let twice = \(n\) {
    return n*n;
};
```
Solution 2: Integrated Unit Testing Features

Unit testcases are expressed as isolated execution points.

```javascript
let twice = \(n\) {
    return n*n;
};
```

- Normal execution point
  - ex. twice(1) returns 1.
- Successful test case
  - ex. twice(2) should return 4 and actually returns 4.
- Failed test case
  - ex. twice(3) should return 6 but actually returns 9.
Solution 2: Integrated Unit Testing Features

Unit testcases are expressed as isolated execution points.

```javascript
let twice = \(n\) {
  return n*n;
};
```

- Normal execution point
  - ex. twice(1) returns 1.

- Successful testcase
  - ex. twice(2) should return 4 and actually returns 4.

- Failed testcase
  - ex. twice(3) should return 6 but actually returns 9.
Solution 2: Integrated Unit Testing Features

Unit testcases are expressed as isolated execution points.

```javascript
let twice = (n) => {
  return n * n;
};
```

- Normal execution point
  - ex. `twice(1)` returns 1.

- Successful testcase
  - ex. `twice(2)` should return 4 and actually returns 4.

- Failed testcase
  - ex. `twice(3)` should return 6 but actually returns 9.
Solution 2: Integrated Unit Testing Features

Unit testcases are expressed as isolated execution points.

```javascript
let twice = \(n\) {
  return n*n;
};
```

- **Normal execution point**
  - ex. `twice(1)` returns 1.

- **Successful testcase**
  - ex. `twice(2)` should return 4 and actually returns 4.

- **Failed testcase**
  - ex. `twice(3)` should return 6 but actually returns 9.

- Shortcuts to promote experiment to unit testcase.
  ```javascript
  #+ s(3) -> [1,2];
  ```
Solution 2: Integrated Unit Testing Features

Unit testcases are expressed as isolated execution points.

Normal execution point
- ex. twice(1) returns 1.

Successful testcase
- ex. twice(2) should return 4 and actually returns 4.

Failed testcase
- ex. twice(3) should return 6 but actually returns 9.

Shortcuts to promote experiment to unit testcase.

Employ returned value as the expected value.
Solution 2: Integrated Unit Testing Features

Unit test cases are expressed as isolated execution points.

```javascript
let twice = \(n\) {
    return n*n;
};
```

- **Normal execution point**
  - ex. `twice(1)` returns 1.

- **Successful testcase**
  - ex. `twice(2)` should return 4 and actually returns 4.

- **Failed testcase**
  - ex. `twice(3)` should return 6 but actually returns 9.

- Shortcuts to promote experiment to unit testcase.
  - **Correct**
    - `+ s(3) -> [1,2];`
  - **Incorrect**
    - `- s(3) -> [];`

Employ returned value as the expected value.

Input expected value by hands.
Solution 2: Integrated Unit Testing Features

Unit test cases are expressed as isolated execution points.

```javascript
// Normal execution point
let twice = \(n\) {
  return n*n;
};

// Successful testcase
ex. twice(2) returns 4.

let s = [1, 2];

// Failed testcase
ex. twice(3) should return 6 but actually returns 9.
```

- Shortcuts to promote experiment to unit testcase.
  - Correct
    - `#+ s(3) -> [1, 2];`
  - Partially correct
    - `- s(3) -> [1, 2, 3] || [1, 2];`
  - Incorrect
    - `- s(3) -> [1, 2];`

Employ returned value as the expected value.

Input expected value by hands.
Solution 3: Shortcut to Take Function Call Out From Logs

We can generate small sub problems by taking out function call, which seems to cause the wrong result.

- Even function value can be serialized.

1. Select execution points to inspect.
2. Show history of n, select bindings.
   - ex. choose bindings where n = 0.
3. Select function call and take it out
   - ex. choose cont(1) and copy-paste.
4. Debug the new execution point.
   - Generated execution point is small sub problem.

We can write unit testcases for anonymous functions.
Conclusion
We designed a set of unit testing features, which goes well with live programming:

- Isolated execution points for large programs,
- Unit testing features for sound programs,
- Making sub-problems from runtime information for easier debugging

Q&A and live coding time with Shiranui.