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.

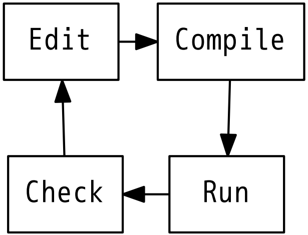
Background: Introduction to Live Programming

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.



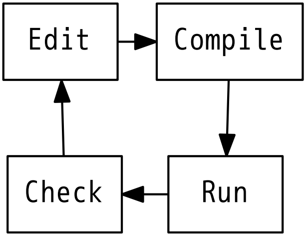
Background: Introduction to Live Programming

Q. What is a live programming environment?

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

Traditional Programming

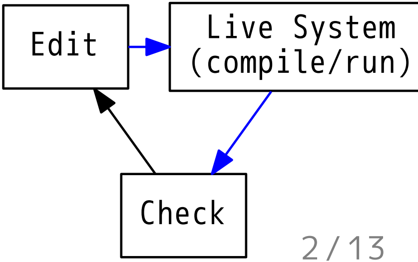
All transitions are manual.



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

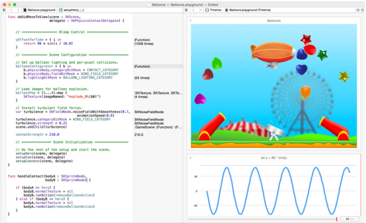
Environments re-evaluate the program when source code or data changes.

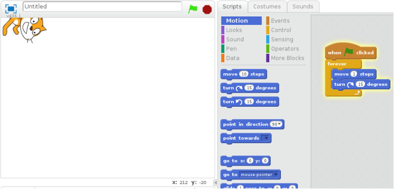
- YinYang [McDirmid, 2013]
- Apple Swift’s Playground
- Our Prototype Shiranui

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?

Question:

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

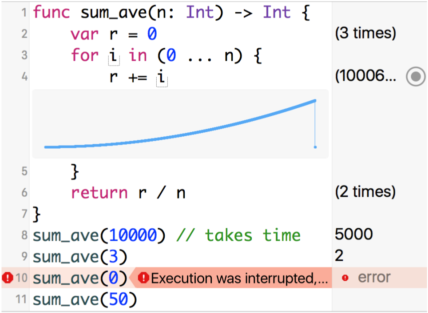
Existing live programming environments have only one entry point.

Problem 1: Single Entry Point

Existing live programming environments have only one entry point.

It's like a big "main" function.

```
void main() {
```



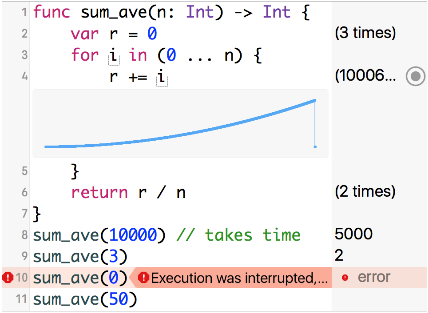

```
}
```

Problem 1: Single Entry Point

Existing live programming environments have only one entry point.

It's like a big "main" function.

```
void main() {
```




```
}
```

=> Not suitable for large programs.

It causes:

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

Problem 1: Single Entry Point

Existing live programming environments have only one entry point.

It's like a big "main" function.

```
void main() {
func sum_ave(n: Int) -> Int {
    var r = 0                           (3 times)
    for i in (0 ... n) {
        r += i                          (10006...
```

```
    }
    return r / n                        (2 times)
}
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,
   - ex. We cannot get `sum_ave(3)`'s feedback before `sum_ave(10000)`.
2. combined runtime log,
3. lost feedback.

Problem 1: Single Entry Point

Existing live programming environments have only one entry point.

It's like a big "main" function.

```
void main() {
func sum_ave(n: Int) -> Int {
    var r = 0                          (3 times)
    for i in (0 ... n) {
        r += i                         (10006...
```

```
    }
    return r / n                       (2 times)
}
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:

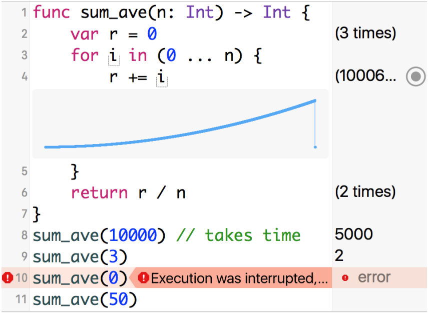
1. long feedback loop,
   - ex. We cannot get `sum_ave(3)`'s feedback before `sum_ave(10000)`.
2. combined runtime log,
   - ex. `sum_ave(10000)`'s log and `sum_ave(3)`'s one are merged.
3. lost feedback.

Problem 1: Single Entry Point

Existing live programming environments have only one entry point.

It's like a big "main" function.

```
void main() {
```




```
}
```

=> Not suitable for large programs.

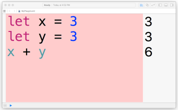
It causes:

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.

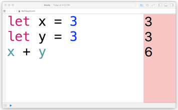
Problem 2: No Support for Testing Frameworks

"Tests" in live programming environments are transient.

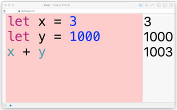
- We need to check all return values by ourselves when we changes source code, because it might change existing functions' behavior.



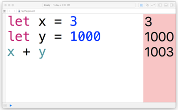
1: coding



2: checking



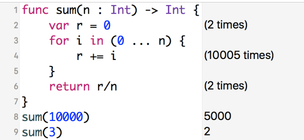
3: coding



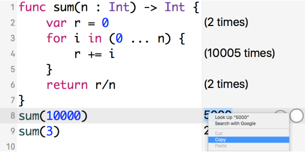
4: checking

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

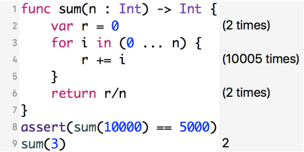
- Constructing expected values is not "live way."
- Promoting experiments to testcases is not supported.



1: check



2: copy value



3: paste value

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 occured?        56
    }else{
        return fact(n-1,                           (10 times)
                    cont:{r in cont(r + n)})       (10 times)
    }                   //incorrect ↑
}
print(fact(10, cont: {r in r}))                    (2 times)
  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

```
#+ sum_ave(10000) -> 5000;
#+ sum_ave(3) -> 2;
#+ sum_ave(0) -> "Division by 0";
#+ sum_ave(5) -> 3;

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  #+ sum_ave(10000) -> 5000;
2  #+ sum_ave(3) -> 2;
3  #+ sum_ave(0) -> "Division by 0";
4  #+ sum_ave(5) -> 3;
5
6  let sum_ave = \(n){
7      let r = ref 0;
8      for i in [1..n]{
9          #* !r -> 0,1,3,6,10;
10         r <- !r + i;
11     }
12     return !r / n;
13 };
```

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

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

```
let sum_ave = \(n){
  // copied body
};
sum_ave(3);
```

```
let sum_ave = \(n){
  // copied body
};
sum_ave(0);
```

```
let sum_ave = \(n){
  // copied body
};
sum_ave(5);
```

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 #+ sum_ave(10000) -> 5000;
2 #+ sum_ave(3) -> 2;
3 #+ sum_ave(0) -> "Division by 0";
4 #+ sum_ave(5) -> 3;
5
6 let sum_ave = \(n){
7     let r = ref 0;
8     for i in [1..n]{
9         #* !r -> 0,1,3,6,10;
10        r <- !r + i;
11    }
12    return !r / n;
13 };
```

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

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

```
let sum_ave = \(n){
  // copied body
};
sum_ave(3);
```

```
let sum_ave = \(n){
  // copied body
};
sum_ave(0);
```

```
let sum_ave = \(n){
  // copied body
};
sum_ave(5);
```

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  #+ sum_ave(10000) -> 5000;
2  #+ sum_ave(3) -> 2;
3  #+ sum_ave(0) -> "Division by 0";
4  #+ sum_ave(5) -> 3;
5
6  let sum_ave = \(n){
7      let r = ref 0;
8      for i in [1..n]{
9          #* !r -> 0,1,3,6,10;
10         r <- !r + i;
11     }
12     return !r / n;
13 };
```

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.

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

```
let sum_ave = \(n){
  // copied body
};
sum_ave(3);
```

```
let sum_ave = \(n){
  // copied body
};
sum_ave(0);
```

```
let sum_ave = \(n){
  // copied body
};
sum_ave(5);
```

Solution 2: Integrated Unit Testing Features

Unit testcases are expressed as isolated execution points.

```
#+ twice(1) -> 1;
#- twice(2) -> 4;
#- twice(3) -> 6 || 9;

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.

```
#+ twice(1) -> 1;
#- twice(2) -> 4;
#- twice(3) -> 6 || 9;

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.

```
#+ twice(1) -> 1;
#- twice(2) -> 4;
#- twice(3) -> 6 || 9;

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.

```
#+ twice(1) -> 1;
#- twice(2) -> 4;
#- twice(3) -> 6 || 9;

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.

```
#+ twice(1) -> 1;
#- twice(2) -> 4;
#- twice(3) -> 6 || 9;

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.

```
#+ s(3) -> [1,2];
```

Solution 2: Integrated Unit Testing Features

Unit testcases are expressed as isolated execution points.

```
#+ twice(1) -> 1;
#- twice(2) -> 4;
#- twice(3) -> 6 || 9;

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.

```
#+ s(3) -> [1,2];   --correct-->   #- s(3) -> [1,2];
```

Employ returned value as the expected value.

Solution 2: Integrated Unit Testing Features

Unit testcases are expressed as isolated execution points.

```
#+ twice(1) -> 1;
#- twice(2) -> 4;
#- twice(3) -> 6 || 9;

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.

`#+ s(3) -> [1,2];`

correct → `#- s(3) -> [1,2];`

Employ returned value as the expected value.

incorrect → `#- s(3) -> ;`

Input expected value by hands.

Solution 2: Integrated Unit Testing Features

Unit testcases are expressed as isolated execution points.

```
#+ twice(1) -> 1;
#- twice(2) -> 4;
#- twice(3) -> 6 || 9;

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.

```
#+ s(3) -> [1,2];
```

correct

```
#- s(3) -> [1,2];
```

Employ returned value as the expected value.

partially correct

```
#- s(3) -> [1,2,3] || [1,2];
```

Modify the parts of returned value.

incorrect

```
#- s(3) -> ;
```

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.

```
#+ fact(2,id) -> 4;
let fact = \fact(n, cont){
  #* n -> 2,1,0;
  if n = 0 {
    return cont(1);
  }
  return fact(n-1, \re(r){
    return cont(n + r);
  });
};
```

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.

```
#+ <|$(cont->$(cont->$()id,n->2)re,n->1)re|>(1) -> 4;
```

We can write unit testcases for anonymous functions.

Conclusion. Questions and Live Coding Time.

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.