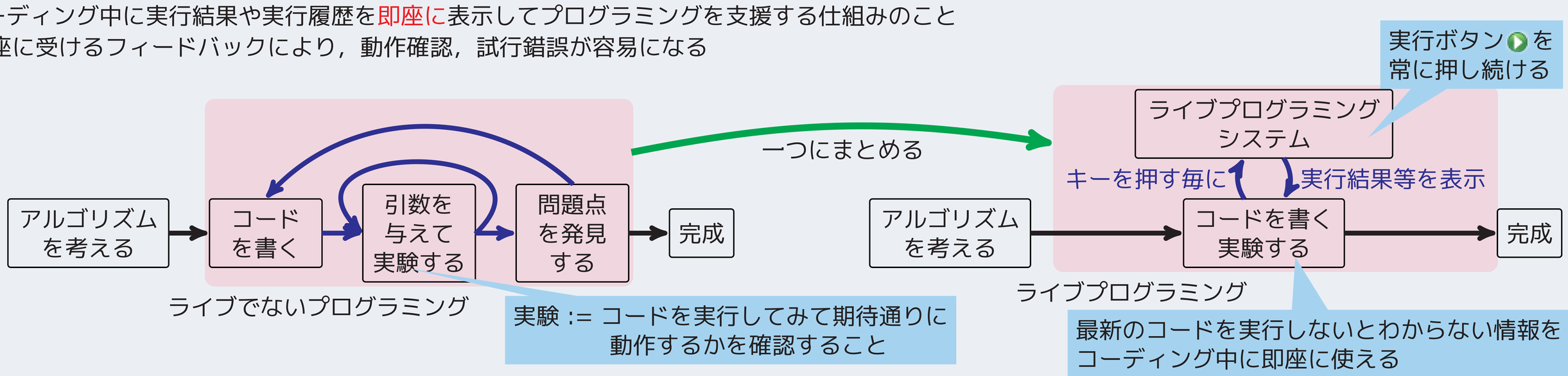
ライブプログラミングにユニットテストを統合する機能の提案

今井 朝貴　増原英彦　青谷 知幸　(東京工業大学)

背景: ライブプログラミングとは

コーディング中に実行結果や実行履歴を即座に表示してプログラミングを支援する仕組みのこと

- 即座に受けるフィードバックにより，動作確認，試行錯誤が容易になる



目的: ライブプログラミングを実用的に

- ユニットテストの統合
- フィードバックの表現能力の向上
- フィードバックの速度の向上

本研究はここに注目!

問題: ユニットテストは作成が大変

- 関数の引数や期待する値を手動で記述するのは大変

```
assertEquals(sum({1,2,3}),6);
assertEquals(sum({1,2,3,4,5}),15);
```

- トップレベルにない関数はテスト，実験しづらい

```
void filter_greater_than_n(int n){
  Predicate<Integer> is_greater_than_n
                     = (Integer k) -> k > n;
  print(is_greater_than_n(readInt())?"Big!":"Small!");
}
```

自由変数nを含む関数オブジェクト

アイデア: ライブプログラミングでユニットテストを楽に作成

- ライブプログラミングの利点
  - 実験，試行錯誤が容易
  - 実行時情報の使用が容易
- **提案手法**
  - **目視で正しいと確認した実験をテストに変換**
  - **計算途中の式の評価結果をテストに使用**

提案1: FlyLine: 実験とテストの統合インターフェイス

```
1  #+ f(1,id) -> 1;
2  #- f(1,id) -> 1;
3  #- f(3,id) -> 6 || -123;
4
5  // f works only when n = 1
6  let f = \fi(n,cont){
7      if n = 1 {
8          return cont(1);
9      }
10     return cont(-123);
11 };
```

- 実験とテストを同じように記述
  - 実験
  - 成功したテスト
  - 失敗したテスト
- 実験からテストに簡単に変換

```
#+ f(1,id) -> 1;
```

正しい → `#- f(1,id) -> 1;` 実験結果を期待する値とする

間違い → `#- f(1,id) -> ;` 手動で期待する値を入力する

提案2: 計算途中の関数呼び出しをテストとして取り出す

```
1  #+ f(2,id) -> 2;
2  let f = \fi(n,cont){
3      #* n -> 2,1;
4      if n = 1 {
5          return cont(1);
6      }
7      return f(n-1,\re(r){
8          return cont(n*r);
9      });
10 };
11 #+ <|a=$(f->a)fi|>(1,<|$(cont->$()i,n->2)re|>) -> 2;
```

- デバッグ対象のFlyLineを選択
- 変数履歴表示，変数環境を選択
- 関数呼び出しを選択，その関数呼び出しを左辺としたFlyLineを生成
- 計算途中の値を使用する利点
  - 複雑な値を手動で書く必要がない
  - 返り値を見てから動作が怪しいものをテストにできる
  - ラムダ式の自由変数の値がわかる

例: 継続渡しスタイルの階乗計算の作成

1. いくつか実験を記述しておく(4~6行目)

```
1 let id = \i(n){
2     return n;
3 };
4 #+ f(1,id) -> "No such variable: f";
5 #+ f(3,id) -> "No such variable: f";
6 #+ f(4,id) -> "No such variable: f";
```

2. 関数fを書く(8~15行目)
実験の右辺は自動的に更新される(4~6行目)

```
4 #+ f(1,id) -> 1;
5 #+ f(3,id) -> 6;
6 #+ f(4,id) -> 10;
7
8 let f = \fi(n,cont){
9     if n = 1 {
10        return cont(1);
11    }
12    return f(n-1,\re(r){
13        return cont(n+r); //bug!
14    });
15 };
```

!!バグ!! 正しくは cont(n*r)

3. 実験をテストに変換する
- 実験結果が正しい(4,5行目)
  → 実験結果を期待する値とする
- 実験結果が正しくない(6行目)
  → 手動で期待する値を入力する

```
4 #- f(1,id) -> 1;
5 #- f(3,id) -> 6;
6 #- f(4,id) -> 24|| 10;
```

4. 失敗したテストを選択する(6行目)
引数の履歴を表示する(9行目)

```
6 #- f(4,id) -> 24 || 10;
7
8 let f = \fi(n,cont){
9     #* n -> 4,3,2,1;
```

5. n=1のときのcont(1)を選択する(11行目)

```
9     #* n -> 4,3,2,1;
10    if n = 1 {
11        return cont(1);
```

6. テストとして取り出し選択する(7行目)
nとrの履歴を表示する(14,15行目)

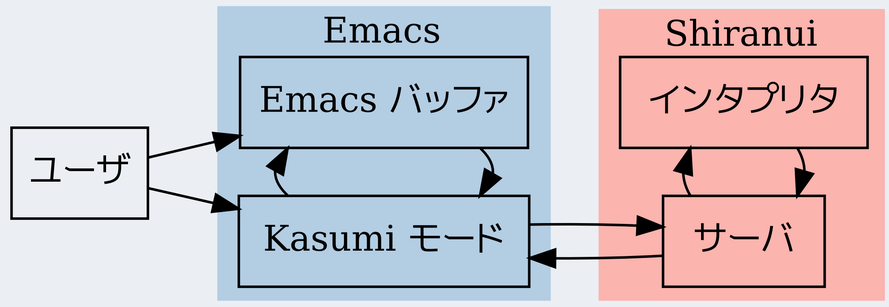
```
7 #+ <|$(cont->$(cont->$(cont->$()i,n->4)re,n->3)re,n->2)re|>(1) -> 10;
13    return f(n-1,\re(r){
14        #* n -> 2,3,4;
15        #* r -> 1,3,6;
16        return cont(n+r); //bug!
17    });
```

7. contの形からcont(n+r)が怪しいと推測可能．cont(n*r)に変更する(17行目)
すると，テストがすべて成功する(4~6行目)

```
4 #- f(1,id) -> 1;
5 #- f(3,id) -> 6;
6 #- f(4,id) -> 24;
14    return f(n-1,\re(r){
15        #* n -> 2,3,4;
16        #* r -> 1,2,6;
17        return cont(n*r); //bug!
18    });
```

プロトタイプ: Shiranui言語環境

- **提案手法を実現するプロトタイプShiranuiを作成，公開**
  - インターフェイスはEmacs上に作成(900行)
  - 独自言語インタプリタ，サーバはC++で作成(7200行)
  - https://github.com/tomoki/Shiranui/
- インタプリタによる実行中に実行時情報の保存
  - 式の評価結果，関数呼び出しの順番
  - 変更可能なオブジェクトの変更履歴
- Shiranui言語には，データを文字列化するためのDSLがある
  - 自由変数を含む関数オブジェクト
  - 共有，循環を含むデータ



今後の課題

- 再実行の高速化
  - → 再実行が必要な実験，テストのみ再実行
- より充実した言語環境
  - 複雑なデータ構造の実装と可視化のサポート
  - 実行時情報からの型推論
- 提案手法を既存言語の開発環境上に実現
  - → 逆向き実行ができるデバッガを利用
  - → 式の評価結果を保存するようなコード変換
- 提案手法により，従来のライブプログラミングシステムを拡張
  - 従来のライブプログラミング言語はグラフィックスを描くものが多い
  - 問題: コードとその出力(絵)を一組しか持たない → 比較実験しづらい
    (例: 線の太さを色々変えて比較)
  - 解決: 提案手法は出力を複数持つ → 比較実験をしながらコードを書ける