Scala Debugger

Serena Chen & Peter Seger

Problem

We want to create a debugger that steps through a program and shows what is happening at each step.

Big Step vs. Small Step Interpretation

Over the course of the semester, we implemented **big step interpretation**:

```
    (let ((a (+ 10 20)) (b 40)) (if (= a 30) a b))
        → 30
```

We changed the implementation to use **small step interpretation**:

```
    (let ((a (+ 10 20)) (b 40)) (if (= a 30) a b))
        → (let ((a 30) (b 40)) (if (= a 30) a b))
            → (let ((a 30) (b 40)) (if true a b))
                    → (let ((a 30) (b 40)) a)
                         → 30
```

This allows us to analyze exactly how the interpreter is breaking down the code.

Class Implementation

- `isBasic`
  - Returns a boolean based off the status of whether the body is basic (completely evaluated)
  - Examples of basic expressions: `EInteger`, `EBoolean`
  - Examples of not basic expressions: `EIf`, `EApply`
- `immValue`
  - Returns the actual value of the body of a basic expression
  - Errors if expression is not basic
- `eval`
  - Recurses with the current body evaluated in the current environment by creating a new “sub-expression”
  - Catches exceptions

Expression Example

```
abstract class Exp {
    def isBasic () : Boolean = false

    def eval (env : Env[Value]) : Result

    def immValue (env : Env[Value]) : Value = {
        error("Not a basic type")
    }
}
```

```
case class EThrow (val e : Exp) extends Exp {
    override def isBasic () : Boolean = {
        return e.isBasic()
    }

    override def immValue (env : Env[Value]) : Value = {
        if (isBasic()) {
            return e.immValue(env)
        }
        error("Thrown value is not fully simplified")
    }

    def eval (env : Env[Value]) : Result = {
        if (e.isBasic()) {
            return new RValue(e.immValue(env));
        } else {
            val new_ec = e.eval(env)
            new_ec match {
                case RException(_) => return new_ec
                case RExp(e) =>return new RExp(new EThrow(e))
            }
        }
    }
}
```

Debug Mode

We implemented two styles of debuggers:

1. Showing the evaluation flow for each step of execution
    a. Shows the abstract representation being made at each step of evaluation
2. Showing how the expression changes for each step of execution
    a. Shows the “logical” flow of the code in a more human readable format

Debugger Demo

Next Steps

- Stepping over expressions
- Adding breakpoints
- Adding more commands to the debugger
- Accessing the environment - viewing it and modifying it