神奈川大学理学部 情報科学科 馬谷研究室

Visitor パターンを用いた言語処理系に対する並行処理の追加

河原 那夏 202003507

1 はじめに

本研究で扱う Lox 言語の式と文の演算の実装には Visitor **パターン**というデザインパターンが使われており, 演算ごとに評価をするための共通のメソッドをオーバーライドで実装し, 言語の拡張がしやすいものである. そのため, ひとつの演算に並行処理を実装できれば, すべての演算に実装が可能である. また, 並行処理によって処理速度を向上が可能であれば, 実行時間が短く, 拡張のしやすいインタプリタの開発に役立てると考えた.

本研究では複数の文を並行に処理する parallel 文という演算を追加し, 通常通り一つずつ順番に処理をしていく方法よりも処理速度が向上するか, 向上していた場合には実行時間がどれほど短縮していたか比較する.

2 背景

Lox 言語 [1] とは動的型付けで, 変数にはあらゆる型の値を格納することができるシンプルな言語である. そのインタプリタは Java 言語で作られており,Lox コードをまず Scanner **クラス**でトークンに分け, それを Parser **クラス**で木構造のようにノードを繋ぐことで式と文の形にし, できたものを Interpreter **クラス**で評価をする.

また, 評価の実装に使う Visitor **パターン**はそれぞれの式や文の演算ごとに評価を実行する同じメソッドを持つクラスを作り, それをクラスごとにオーバーライドで実装することで適切に評価を行うことができるものである. このパターンを使って演算ごとのクラスを生成スクリプトを使って実装することで演算を追加する作業を減らすことができ, また, 演算の定義をサブクラスで行うことで, 演算の定義を変えるたびに生成スクリプトそのものを書き換える必要が無くなる.

以上を踏まえ,parallel 文の実装をする.

3 parallel 文

複数の文を並行に処理をする parallel 文を追加する.

- Lox コードでの使用例

  文字列を受け取り, それを繰り返し出力する関数を parallel 文を用いて記述すると以下のようになる.

```
fun a(n){
  for (var i = 0; i < 10000000;
   i = i + 1) { print n ; }}
fun paralleltest(){
 parallel{
  a("a"); a("b"); a("c"); a("d");}
 // a("a"); a("b"); a("c"); a("d");}
paralleltest();
```

- 実行結果

  文字列をランダムな順番に出力していることから, 並行に処理されていることが確認できる.

```
d c a b c d c c a b a c d c a …
```

4 実装

実際にこの parallel 文を実装する.

- TokenType.java

  まず Scanner **クラス**が Lox コードに書かれる parallel 文のキーワード「parallel」をトークンにするために TokenType に PARALLEL トークンを追加する.

```
enum TokenType { …
PRINT, PARALLEL, RETURN, SUPER, THIS,
TRUE, … EOF }
```

- Scanner.java

  Scanner **クラス**が parallel を PARALLEL トークンにするように追加する.

```
keywords.put("parallel",  PARALLEL);
```

- Parser.java

  Parser **クラス**が PARALLEL トークンを見つけると parallelStatement() を行う. parallel 文の中にある文をリストにし, それをフィールドに入れた PARALLEL **クラス**のインスタンスを返す.

```
private Stmt parallelStatement(){
 List<Stmt> bodies
  = new ArrayList<Stmt>();
 consume(LEFT_BRACE,
```

```
 "Expect '{' after 'parallel'.");
while(!check(RIGHT_BRACE)){
 … //要素を 255 まで受け取る
   bodies.add(statement()); }
consume(RIGHT_BRACE,
 /*エラーメッセージ*/);
return new Stmt.Parallel(bodies); }
```

- GenerateAst.java
  このコードでは追加したい演算の名前とそのフィールドを追加することで Stmt.java に対応したクラスを生成する.

```
"Parallel   : List<Stmt> stmtS"
```

- Stmt.java
  「GenerateAst.java」より Visitor **パターン**のためのクラスが生成されている.

```
static class Parallel extends Stmt {
 Parallel(List<Stmt> stmtS) {
  this.stmtS = stmtS; }
@Override
<R> R accept(Visitor<R> visitor) {
 return
  visitor.visitParallelStmt(this);
} final List<Stmt> stmtS; }
```

- Interpreter.java (visit メソッド)
  Interpreter **クラス**では変数の値などの状態を持っており, また,Interpreter **クラス**はブロックごとに呼び出されているためブロックごとに状態がある. visitParallelStmt() では並行に処理する文ごとに ForkJoin を持つクラスのインスタンスを割り当てる. 引数には文ひとつと Interpreter のインスタンスを複製したものを入れ,fork(),join() で実行している. 複製したインスタンスは parallel 内の状態だけを持っているので, 外側の状態は parallel 内のすべての文で共有している.

```
@Override
public Void
 visitParallelStmt
    (Stmt.Parallel stmt){
  int size = stmt.stmtS.size();
  List<ExeFork> exeforks
   = new ArrayList<ExeFork>();
  for(int i = 0; i < size - 1; i++){
   exeforks.add(new ExeFork
    (stmt.stmtS.get(i),
            this.clone()));
    exeforks.get(i).fork(); }
  execute(stmt.stmtS.get(size - 1));
  for(int i = 0; i < size - 1; i++){
    exeforks.get(i).join();
  } return null; }
```

- Interpreter.java (ForkJoin)
  受け取った文を複製したインスタンスを使い評価する.

```
class ExeFork
        extends ForkJoinTask<Object>{
 Stmt stmt;
 Interpreter interpreter;
 Object result;
 ExeFork(Stmt s, Interpreter i){
 stmt = s;
 interpreter = i; }
…
 public boolean exec(){
  setRawResult
    (stmt.accept(interpreter));
  return true; }}
```

5 評価

実際に追加した parallel 文を使用する場合と使用しない場合で実行時間を比較する. 比較する Lox コードは「Lox コードでの使用例」のものを使うが, 文字の表示を行わずに比較する. 測定には System.currentTimeMillis() を使い,main メソッド内で runFile() を 10 回行い, それをまた 10 回実行することで計 100 回 Lox コードを評価し, その 1 回あたりの平均時間を実行時間とする.

- parallel 文を使った場合
  平均 2739.24ms
- parallel 文を使わなかった場合
  平均 4711.2ms

parallel 文を使った場合, 使わなかった場合に比べて 0.58 倍の実行時間と処理速度が向上した.

6 まとめ

本研究では複数の文を並行に処理する parallel 文を実装し,parallel 文を使わなかった場合よりも実行時間を短くすることができ,Visitor **パターン**を用いたインタプリタに対し, 並行処理の有効性を示すことができた.

参考文献

[1] Robert Nystrom, Crafting Interpreters, https://craftinginterpreters.com/.