関数型プログラミング

DbLab141f

d8161105 渡部未来

関数型プログラミング

関数型プログラミングは、宣言型とも言われ、
問題の性質を記述するプログラミングパラダイムである。

関数型プログラミング　　命令形プログラミング

問題の性質を記述　　問題の解き方を記述

```
def sumSales(salesList: Seq[Sales]): Int =
              // 必要な要素だけをフィルタリングする。
    salesList.filter(_.productId == 1)
              // 畳み込み演算により、合計を求める
             .foldLeft(0){_ + _.count}
```

```
SELECT SUM(count) FROM sales WHERE product_id = 1
```

```
int sum_sales(Sales *sales_list, int sales_length) {
    // 合計を表す変数を宣言，0で初期化
    int sum = 0;
    // ループを用いて配列の0番目から末尾を走査していく。
    for(int i=0; i<sales_length; i++) {
        // if文による条件分岐
        if (sales_list[i].product_id == 1) {
            // 合計を表す変数に、値を足しこんでいく。
            sum += sales_list[i].count;
        }
    }
    return sum;
}
```

参考: 関数型プログラミングとは結局何なのか
http://blog.kokuyouwind.com/archives/808

関数型プログラミング

主にサポートされている機能

- 関数は第一級関数(ファーストクラスファンクション)
- 強力な型システム(型推論)
- 参照透過
- 高階関数
- 再帰
- 遅延評価
- 制御構文ではなく制御式

関数型プログラミング

禁止する項目

- 副作用の禁止
  - 変数への再代入(forやwhileの禁止)
  - データ構造を直接変更する
  - (命令型のアプローチによる)標準入出力、ファイルへの入出力などの禁止

関数型プログラミング言語

言語	純粋関数型	関数型	非関数型
説明	前頁副作用を完全に排除し、積極的に関数型のための機能を取りれている言語。	副作用を許可しつつも、関数型プログラミングの機能をサポートし、推奨する言語。	命令型、もしくは別のプログラミングスタイルを推奨する。ただし、関数型プログラミングができないわけではない。
具体例	Haskell, Clean, Idris, Mirandra	Scala, Ocaml, F#, LISP	JavaScript, Java, C++

関数型プログラミングによるメリット

入力した列を逆順で出力する例

```
1 val input = StdIn.readLine().split(" ")
2 val output = input.reverse // inputは値を変化させない！
3 println(output.mkString(" "))
```

- 抽象的な関数が多く用意されている
- 抽象的な関数を多く利用することで、可読性が上昇、短いプログラムで済む
- 手続き処理を含まないためバグが混入しにくい

```
 1 // 入力のためのfor
 2 for(i=0; i<N; i++){
 3   scanf("%d", &input[i]);
 4 }
 5 // 計算のためのfor
 6 for(i=0; i<N; i++){
 7   output[i] = input[N-i-1];
 8 }
 9 // 出力のためのfor
10 for(i=0 ;i<N ; i++){
11   printf("%d ", output[i]);
12 }
```

- すべての処理を具体的に処理しないといけないため、可読性が低い
- プログラムが長く冗長的になってしまう

関数型プログラミングによるメリット

モジュラリティが高く、テストが行いやすい

```
def foo(x: Int):Int =
  // if文ではなく式なので、値を返す
  if(x % 2 == 0) x * 3 else x * 2 + 1

assert(foo(2) == 6)
assert(foo(3) == 7)
assert(foo(3) == 7)
```

参照透過が保たれていれば(関数の出力が入力のみに依存する)推論がしやすい。
テストも容易である。

```
int x = 1

int bar(int y){
   int output = x * 2 + y;
   x = y; // グローバル変数yを更新
   return output;
}

assert(bar(2) == 4)
assert(bar(2) == 4)
// assert failed bar(2) == 6
```

```
int hoge(int y){
   int x;
   scanf("%d", &x); // 標準入力を受け付ける
   return x * 2 + y;
}

assert(hoge(5) == unknown_value)
// 不定の値
```

副作用がある場合(グローバル変数の参照や配列、標準入出力) 関数の入力に対して出力が不定のためテストが難しい。

関数型プログラミングによるメリット

モジュラリティが高く、組み合わせが行いやすい

```
1 val double = (x: Int) => x * 2
2 val plusThree = (x: Int) => x + 3
3 val minusOne = (x: Int) => x - 1
4 val doublePlusThree = double compose plusThree
5 val plusThreeMinusOne = plusThree compose minusOne
6 val all = double compose plusThree compose minusOne
7
8 doublePlusThree(3) // double(PlusThree(3))
9 => 12
10 plusThreeMinusOne(3) // plusThree(minusOne(3))
11 => 5
12 all(3)  // double(plusThree(minusOne(3)))
13 => 10
14 (1 to 10).map(all)
15 => Vector(6, 8, 10, 12, 14, 16, 18, 20, 22, 24)
```

関数が値として扱える

それぞれの関数がモジュラリティが高いため関数合成ができる。また、単体テストにより各関数の動作が保証されている場合には、合成をおこなったあとの関数の動作も保証されやすい。

関数型プログラミングによるメリット

状態(文脈)を型で表すことで値として処理できる

```scala
def doubleEven(x: Int): Option[Int] =
  if(x % 2 == 0) Some(x * 2) else None

doubleEven(5) match {
  // 全ての取りうる状態を場合分けしなければ
  // コンパイル時に警告を出す。
  case Some(x) => x + 1
  case None => 0
}
=> 0
// 上と同じ意味エレガントな書き方
doubleEven(6).map((x:Int) => x + 1).getOrElse(0)
=> 13
// 更にエレガント
doubleEven(6).map(_ + 1).getOrElse(0)
```

値が存在する(Some)かしない(None)かを表すOption型なのでシグネチャを見ることで、どのような関数なのか把握できる。適切な処理(パターンマッチ・高階関数呼び出し等)をしなければコンパイラがエラー・警告を出して知らせてくれる。

```java
    public Integer doubleEven(int x){
        if(x % 2 == 0) return x * 2;
        else return null;
    }

    public Test(){
        // NullPointerException ifでnull分岐が必要。
        // コンパイラ時には警告は出さない。
        doubleEven(5) + 1;

        Integer result = doubleEven(5);
        if(result != null){
            result *= 2;
        }else{
            result = 0;
        }
    }
```

値の状態をnullであるかそうでないかを使用者が判断しなければならないため、ドキュメントが必須になる。また、適切なnull処置をしていても、していなくてもコンパイラが関与しないため、実行時の障害となる。(もし、ミッションクリティカルだった場合は・・・?)

関数型プログラミングによるデメリット

- 宣言的な記述のため、知識がないとパフォーマンスチューニングが難しい
- コンパイルが重くなりがち
- IOなど、本質的に副作用のある処理の記述が難しい

パフォーマンスに関しては、近年のマシンスペックの向上やコンパイラの最適化技術の向上により、シビアなパフォーマンスを求められる分野で無ければ問題が少ない。また、宣言的なプログラミングはコンピュータ寄りの命令形に比べ人間に近く可読性が高い、バグが混入しにくいなどの利点が多いため、あまりデメリットではない。

命令形から関数型への移行

関数型は命令型スタイルのプログラミングに慣れていると、思考の切り替えが難しい。しかし、徐々に思考を切り替えていくことで関数型の恩恵を確実に得ることができる。

- const・finalキーワードを基本的に変数に付け、再代入を減らす
- 再帰呼び出しを利用する
- 副作用がない関数作りを心がける(グローバル変数, 配列の使用, 再代入を控える)
- 制御構文ではなく制御式が導入してある言語を使用する(Ruby, Scala, 純粋関数型言語)
- 配列を避け、ListやVectorを利用する
- (使用している言語がサポートしていれば)高階関数を利用して、List・Vectorを操作する(Java8, JavaScript, Ruby, Swiftは関数型ではないが導入済み)
- 状態を型で管理する(OptionalはJava8導入済み, オブジェクト指向言語であれば工夫次第で状態を型で管理できる)
- 純粋ではない関数型言語を使用する
- 純粋関数型言語を使用する

まとめ

- 宣言的に記述するプログラミング手法を関数型プログラミングと呼ぶ
- 関数型には制約(副作用の禁止)が存在する
- 制約を守るために様々な機能がサポートされている
- 制約を守ることで、多くの恩恵(直感的な記述・短いプログラム・バグの大きな軽減・コンパイルサポートが得れるエラー機構)を得ることができる
- 段階的に関数型に移行することで、確実に恩恵を得ることができる