oapatternmatching

No Author Given

No Institute Given

Abstract. Abstract

1 Introduction

Introduction

2 Problem

We have seen a lot of cool usage of Object Algebras[], and the extensibility of OAs allows us to build extensible and composible systems when dealing with recursive data types. But sometimes for some definitions we need to do pattern matching (examples will be given later). Using pattern matching is convenient, if not provided, there are still ways to do it[], but then it would be very complicated to define some terms.

For example, in the MDSLCS paper[1], the Domain-Specific Language of Regions are defined with the optimization in the union definition: In the union(reg1,reg2) operator, they want to figure out whether reg1 or reg2 is the universal region, if so, do the optimization: the result goes directly to the universal region, otherwise, do not optimize. Apparently, this optimization relies on pattern matching, and they use case classes in Scala to support this.

Although case classes have much benefit in that they comes with constructors and pattern matching power, the drawback of case classes is obvious: hard to deal with new cases (possibly need to explain more).

Another example is equality function, that checks whether two expressions are syntactically equal or not. This operations needs to traverse the two expressions and compare their components, which apparently needs pattern matching.

The pattern matching power of case classes is what we don't have in OAs. For example, if we write the Region DSL using OA, the language interface would be like:

```
trait RegionAlg[Region] {
  def Univ() : Region
  def Circle(radius : Double) : Region
  def Union(reg1 : Region, reg2 : Region) : Region
}
```

With this region algebra signature, writing compositional interpretations like `EvalRegion` (in MDSLCS[1]) is easy, it can be expressed as a normal algebra. But

for non-compositional interpretations like `OptimizeRegion` that needs pattern matching, it's hard to express using OA.

So the problem we want to solve is to improve OA with pattern matching power so that we can solve problems that needs both extensibility and non-compositional definitions using pattern matching.

3 Solution

(Solution shortly explained with the arithmatic expressions example.)

1. Define `ExpAlg` interface:

```
trait ExpAlg[In, Out] {
  def Lit(x : Int) : Out
  def Add(e1 : In, e2 : In) : Out
}
```

2. For `ExpAlg`, define the corresponding `PatternExpAlg` that extends `ExpAlg` with the type parameter `In` the subtype of `InvExp` (which give us the pattern matching power using Scala's support of Options).

```
trait InvExp[Exp] {
  val fromLit : Option[Int]
  val fromAdd : Option[(Exp, Exp)]
}
trait PatternExpAlg[In <: InvExp[In], Out] extends ExpAlg[In, Out] {
  object Lit { def unapply(e : In) : Option[Int] = e.fromLit }
  object Add { def unapply(e : In) : Option[(In, In)] = e.fromAdd }
}
```

3. For any features that needs pattern matching, simply extending `PatternExpAlg` will give us the pattern matching interface automatically. For example, in the evaluation operation, we could do pattern matching like this:

```
trait EvalExpAlg[In <: InvExp[In] with Eval] extends PatternExpAlg[In,
     Eval] {
  def Lit(x : Int) = new Eval { def eval = x }

  def Add(e1 : In, e2 : In) = new Eval {
    def eval = e1 match {
      case Lit(n) =>
        System.out.println("Been here"); n + e2.eval
      case Add(_, _) =>
        System.out.println("Been there"); e1.eval + e2.eval
      case _      => e1.eval + e2.eval
    }
  }
}
```

4. To construct a concrete expression that supports evaluation and pattern matching, we need to merge the `EvalExpAlg` with the `InvExpAlg` below that really does the pattern matching job:

```
trait InvExpAlg[In] extends ExpAlg[In, InvExp[In]] {
  def Lit(x : Int) = new InvExp[In] {
    val fromLit = Some(x)
    val fromAdd = None
  }
  def Add(e1 : In, e2 : In) = new InvExp[In] {
    val fromLit = None
    val fromAdd = Some(e1, e2)
  }
}
```

The merge operation can be easily defined just for merging `EvalExpAlg` and `InvExpAlg`, for a generic merge, we can use the infrastructure created in FOPwOA paper[2]. (need to explain more).

4 Related Work

References

1. Hofer, C., Ostermann, K.: Modular domain-specific language components in scala. In: GPCE '10 (2010)
2. Oliveira, B.C.d.S., van der Storm, T., Loh, A., Cook, W.R.: Feature-oriented programming with object algebras. In: ECOOP'13 (2013)