Scrap your Boilerplate with Object Algebras!

Author1[1] Author2[2]

[1]The University of Hong Kong
author1@cs.hku.hk
[2] The University of Hong Kong
author2@cs.hku.hk

Abstract. This is the abstract ...

1 Introduction

This is the Introduction.

2 Overview

We start by considering the company structure introduced in Figure 1.

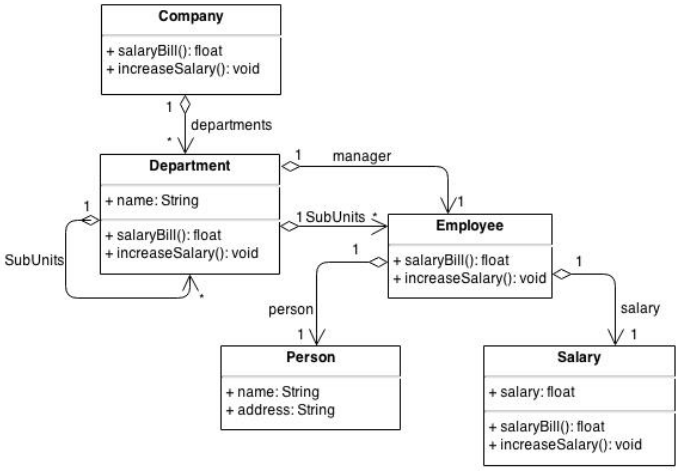

Fig. 1. Company Structure

2.1 Object Oriented Solution

A very natural Object-Oriented way to model the company structure is as illustrated in Figure 2 and Figure 3. Similar code can be applied to Department,

```
public class Company {
  private List<Department> depts;
  public Company(List<Department> depts){this.depts = depts;}
  public Float salaryBill(){
    Float r = 0f;
    for (Department dept: depts) r+= dept.salaryBill();
    return r;
  }
  public void increaseSalary(){
    for (Department dept: depts) dept.increaseSalary();
  }
}
```

Fig. 2. Company Class of OOP style

```
public class Salary {
  private Float salary;
  public Salary(Float salary){this.salary = salary;}
  public Float salaryBill(){return this.salary;}
  public void increaseSalary(){this.salary *= 1.1f;}
}
```

Fig. 3. Salary Class of OOP style

Employee, SubUnit and Person. A Company comprises a list of Departments. Each Department is managed by an Employee as the manager and contains a list of SubUnits. The SubUnit can be either a department or an Employee. An Employee is a Person with the Salary Information.

Now consider adding two operations to our company structure: query the salary bill for the whole company and increase the salary of each employee by 10%. The methods Float salaryBill() and void increaseSalary() in Figure 2 and Figure 3 is an easy solution.

This way of Object Oriented style representation of tree structures can become cumbersome and inflexible due to the bound relationship between classes. For instance, adding a new operation such as pretty printing of the company structure requires a lot of changes on the existing code and violates the no modification rule.

2.2 Modeling Company Structure with Object Algebras

Object Algebras is a good solution to solve the extensibility problem. Figure 4 shows the approach to model the Company structure with Object Algebras.

Hence different operations can be realized by inheriting object algebras from the object algebra interface. To implement query bill operation for the whole Company structure, we need to implement the Company interface with specific operation for each component.

```
public class SalaryQuerySybAlg implements SybAlg<Float,Float,Float,Float,
    Float,Float> {
```

```
@Algebra
public interface SybAlg<Company, Dept, SubUnit, Employee, Person, Salary>{
  public Company C(List<Dept> depts);
  public Dept D(String name, Employee manager, List<SubUnit> subUnits);
  public SubUnit PU(Employee employee);
  public SubUnit DU(Dept dept);
  public Employee E(Person p, Salary s);
  public Person P(String name, String address);
  public Salary S(float salary);
}
```

Fig. 4. Company Structure represented by Object Algebra Interface

```
  public Float C(List<Float> depts){
    Float r = 0f;
    for (Float f: depts) r += f;
    return r;
  }
  ...
  public Float S(float salary){
    return salary;
  }
}
```

While IncreaseSalary can be realized as:

```
public class IncreaseSalarySybAlg implements SybAlg<Float, Float, Float,
    Float, Float, Float> {
  public SybAlg<Float, Float, Float, Float, Float, Float> alg;
  public IncreaseSalarySybAlg(SybAlg<Float, Float, Float, Float, Float,
      Float> alg) { this.alg = alg; }
  public Float C(List<Float> depts) {
    return alg.C(depts);
  }
  ...
  public Float S(float salary) {
    return alg.S(salary*1.1f);
  }
}
```

An increase salary algebra is used to raise the salary for each employee based on a given algebra.

However, although we solved the problem of extensibility with object algebras, the traversal code become so long and most of the time we are writing boilerplate routine code, which is to call the methods of its child leaves. The only code we are really interested in is the Salary S(Float salary) method to return or to increase the salary. It will be great if the boilerplate code can be generated automatically every time we want to traverse the tree structure.

```
public class FloatQuery extends SybAlgQuery<Float> {
  public FloatQuery(Monoid<Float> m) {super(m);}
  public Float S(float p0) {return p0;}
}
```

Fig. 5. Query Salary Class with Object Algebra Framework

```
public class IncSalary extends SybAlgTransform<Float, Float, Float, Float,
    Float, Float> {
  public IncSalary(SybAlg<Float, Float, Float, Float, Float, Float> alg) {
      super(alg);}
  public Float S(float salary) {return alg.S(1.1f * salary);}
}
```

Fig. 6. Increase Salary Class with Object Algebra Framework

2.3 Object Algebra Framework

Motivated by this problem of writing generic code for tree structure traversals, we introduce generic queries and transformations for Object Algebras, which can be easily inherited by real cases of queries and transformations. Furthermore, we designed an object algebra framework with great features. With our framework, even the generic query and transformation code can be generated automatically by adding an "@Algebra" annotation.

Now with our Object Algebra Framework, the code we need to write for Salary Bill and Increase Salary will be much shorter. A Generic query code will be as short as Figure 5. Transformation code will be like Figure 6 The classes SybAlgQuery<R>, SybAlgTransform<R,R,R,R,R,R> are generated by the framework automatically.

3 Queries

As a specific type of object algebras, queries allow users to define new operations handling a user-defined data structure. A *query algebra* is a class implementing an object algebra interface by a top-down traversal throughout the hierarchy. It is something supporting the program to gather information from the substructures of a data type recursively, and make a response at the root node to the query.

3.1 FreeVars: a simple query algebra

An example is shown here to discuss about query algebras in a clearer way. The object algebra interface is related to an expression, where it can be treated as a literal, a string or composed of two smaller ones. Specifically, the structure is defined as follows:

```
public interface ExpAlg<Exp> {
   Exp Var(String s);
   Exp Lit(int i);
   Exp Add(Exp e1, Exp e2);
}
```

Based on the interface above, a query might be raised on collecting all the names of free variables defined in an expression. More precisely, an array of strings would be used to store the names of those variables. In that case, a $Var(s)$ would simply return an one-element array of s, and a $Lit(i)$ corresponds to an empty set, whereas two arrays would be merged into one if we are combining two expressions with the $Add()$ method.

Generally speaking, it is natural to deal with the traversal in an algebra-based approach like this:

```
public interface FreeVarsExpAlg extends ExpAlg<String[]> {
   default String[] Var(String s) {
      return new String[]{s};
   }
   default String[] Lit(int i) {
      return new String[]{};
   }
   default String[] Add(String[] e1, String[] e2) {
      int e1len = e1.length;
      int e2len = e2.length;
      String[] res = new String[e1len+e2len];
      System.arraycopy(e1, 0, res, 0, e1len);
      System.arraycopy(e2, 0, res, e1len, e2len);
      return res;
   }
}
```

Information on our query is collected by traversal and passed on to a higher-level structure. Nonetheless, a programmer has to write a lot of boring code handling the traversals, and it could be even worse for a more complicated data structure. Moreover, it is a query-based approach: you still have to write a bunch of similar stuff with a different query raised, for instance, a pretty printer.

3.2 Generic query algebra with a monoid

Queries are so similar actually: a user has to indicate the rules in which the program may address cases on primitive types and "append" the information. With these two issues, everything becomes simple in the traversal. Hence we introduce the concept of monoid and generic traversal here in our query algebras.

```
public interface Monoid<R> {
   R join(R x, R y);
   R empty();
   default R fold(List<R> lr){
    R res = empty();
```

```
    for (R r: lr){
      res = join(res, r);
    }
    return res;
  }
}
```

The interface of a monoid is defined above. Intuitively, the join() method implies how we gather the information from substructures during merging, and the empty() is just an indicator of "no information". Hence now we are able to write a "generic traversal" manually based on monoids as follows:

```
public class QueryExpAlg<Exp> implements ExpAlg<Exp> {
 private Monoid<Exp> m;
 public Monoid<Exp> m() { return m; }
 public QueryExpAlg(Monoid<Exp> m) {
  this.m = m;
 }
 public Exp Add(Exp p0,Exp p1) {
  Exp res = m.empty();
  res = m.join(res, p0);
  res = m.join(res, p1);
  return res;
 }
 public Exp Lit(int p0) {
  Exp res = m.empty();
  return res;
 }
 public Exp Var(String p0) {
  Exp res = m.empty();
  return res;
 }
}
```

Fig. 7. Generic Query by hand with Monoid

And now we find everything goes in an easier way: we don't care about what kind of query it is any more during the traversal. Despite whether it asks for all the names of free variables or a printer showing the hierarchy of an expression, all we need to do is to traverse a monoid to the generic traversal class, and the monoid is exactly constructed by implementing the previous interface. This is the progress, once we prepare such a template dealing with the traversal, all query algebras can be addressed in a more concise way, which is called the *generic query algebra*.

```
public interface G_Exp {
 <Exp> Exp accept(ExpAlg<Exp> alg);
}
```

Fig. 8. Generic Visitable Interface

3.3 Solving freeVars with generic query algebra

As an alternative way to handle the freeVars query, the query algebra is going to be a subclass of QueryExpAlg, the generic algebra, with generic type to be String[]. To use the generic traversal code, a monoid is defined as follows:

```
public class FreeVarsMonoid implements Monoid<String[]> {
   public String[] empty() {
      return new String[]{};
   }
   public String[] join(String[] e1, String[] e2) {
      int e1len = e1.length;
      int e2len = e2.length;
      String[] res = new String[e1len+e2len];
      System.arraycopy(e1, 0, res, 0, e1len);
      System.arraycopy(e2, 0, res, e1len, e2len);
      return res;
   }
}
```

But the result for an expression can only be a null array based on the monoid. Thus in the freeVars query, furthermore, we expect the variables to store their names into an array, and by using the monoid, the query algebra will be like:

```
public class FreeVarsQueryExpAlg extends QueryExpAlg<String[]> {
   public FreeVarsQueryExpAlg(Monoid<String[]> m) {super(m);}
   public String[] Var(String s) {return new String[]{s};}
}
```

When the class FreeVarsQueryExpAlg is used, an object of the FreeVarsMonoid should be created and traversed to the constructor. As we can see, it is needless for a user to address the traversals in a data structure. Nothing but a monoid is required together with a few methods being overwritten. And furthermore, a monoid can usually be shared among query algebras with the same data type.

4 Transformations

Before writing generic transformation interfaces, we first introduce *generic visitable interface* as Figure 8. Implementing the generic visitable interface allows the user to construct objects from the passed in algebra.

The generic transform interface is constructed by inheriting from the *Object Algebra Interface* with *Generic Visitable Interfaces* as Figure 9. Note that the returned *Generic Visitable Interface* will contain all the information for the tree structure.

```
public interface ExpAlgTransform extends ExpAlg<G_Exp> {
 @Override
 default G_Exp Add(G_Exp e1, G_Exp e2) {
  return new G_Exp() {
   @Override
   public <Exp> Exp accept(ExpAlg<Exp> alg) {
    return alg.Add(e1.accept(alg), e2.accept(alg));
   }
  };
 }
 @Override
 default G_Exp Lit(int i) {
  return new G_Exp() {
   @Override
   public <Exp> Exp accept(ExpAlg<Exp> alg) {
    return alg.Lit(i);
   }
  };
 }
 @Override
 default G_Exp Var(String s) {
  return new G_Exp() {
   @Override
   public <Exp> Exp accept(ExpAlg<Exp> alg) {
    return alg.Var(s);
   }
  };
 }
}
```

Fig. 9. Generic Transformation by hand with acceptor interface

Now to create a specific transformation, e.g., substitute one variable with another name, it can be easily implemented by inheriting from the generic transform interface with the implementation of interesting cases. Here only the method G_Exp Var(String ss) needs to be overridden as Figure 10.

5 Object Algebras Framework

Generic queries and transformations can help users write tree structure traversal code with more extensibility and flexibility. However, writing the generic query and transformation interfaces is still painful experience by itself. It will be even better if these boilerplate code can be generated automatically. If we pay more attention to our generic query and transformation interfaces, without much difficulty we can find that the query and transform code structures for all *Object Algebra Interfaces* are quite similar. Therefore we can make this code generation process automatic.

```
public class SubstVarsTransform implements ExpAlgTransform{
  private String s;
  private G_Exp gExp;
  public SubstVarsTransform(String s, G_Exp gExp){
    this.s = s;
    this.gExp = gExp;
  }
  @Override
  public G_Exp Var(String ss) {
    return new G_Exp() {
      @Override
      public <Exp> Exp accept(ExpAlg<Exp> alg) {
        if (ss.equals(s)) return gExp.accept(alg);
        else return alg.Var(ss);
      }
    };
  }

}
```

Fig. 10. Substitute Variables Transformation

To address this problem, we provide an *Object Algebra Framework*, which utilizes *Java Annotation* to generate generic query and transformation interfaces based on the *Object Algebra Interface*. as illustrated below:

```
@Algebra
public interface ExpAlg<Exp> {
  Exp Var(String s);
  Exp Lit(int i);
  Exp Add(Exp e1, Exp e2);
}
```

With the annotation "@Algebra", the framework will generate the boilerplate codes for us automatically. As for our ExpAlg example, the following directory structure will be generated by the library.

```
src/
├── query/
│   └── ExpAlgQuery
└── transform/
    └── ExpAlgTransform
```

Here the automatically generated ExpAlgQuery, ExpAlgTransform are exactly the same code as we discussed in Section 3 and Section 10. Furthermore, the monoid interface is also included in the *Object Algebra Framework*.

Hence when programming with query, the programmer can skip the intermediate steps such as constructing generic queries and transformations, but only focus on rewriting the interesting cases. For instance, in our ExpAlg example, to implement FreeVars algebra, we can simply override the Exp Var(String s) method of ExpAlgQuery class to return variable name, and provide the specific monoid needed, which in this case will be a String List monoid. While the Sub-

stVars algebra can be realized by overriding the Exp Var(String s) method of ExpAlgTransform interface, which substitutes variable names as specified.

6 Other Features

One More section

7 Case Study

Case study section

8 Related Work

Object Algebras. B. Oliveira and W. Cook proposed Object Algebra as a solution to Expression Problem.

Scrap Your Boilerplate. R. Lammel and S. P. Jones presented a design pattern to traverse data structure of recursive data types in haskell. Such programs usually have much "'boilerplate"' code as we have in our object algebra tree structures.

Visitor Combinators. J. Visser introduced vistor combinators to compose new visitors from given ones.

Almost Compositional Functions.

text

9 Conclusion

And conclusion.

Acknowledgements We should thank someone!

References