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

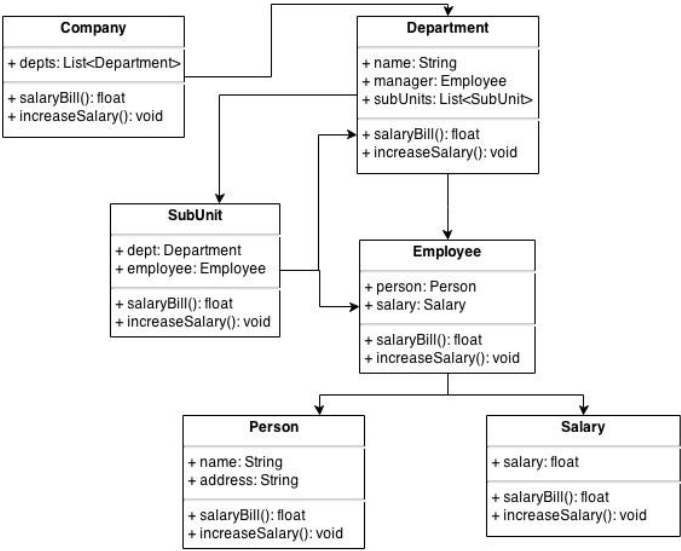We begin by considering the company structure introduced in Figure 1.

Fig. 1. Company Structure

A company is divided into departments which in turn have a manager, and consists of a collection of sub-units. A unit is either a single employee or a

```
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

department. Both managers and ordinary employees are persons receiving a salary. A common OOP programmer may organize the code like Figure 2 and Figure 3. Similar code can be applied to Department, Employee, SubUnit and Person.

Now consider adding two operations to our company structure: query the salary bill for the whole company and increase the salary for the each employee by 10%. The methods Float salaryBill() and void increaseSalary() in Figure 2 and Figure 3 show an easy solution that an OOP programmer will possibly come up with.

This way of OOP style representation of tree structures can become cumbersome and inflexible due to the bound relationship between classes. For example, 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. One can solve this problem by coding with object algebras as Figure 4:

Hence different operations can be realized by inheriting object algebras from object algebra interface. Figure 5 shows how to implement query salary operation based on the company algebra interface.

Increasing Salary is trickier. We need the visitable classes to help reconstruct the tree structures.

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

```
@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

the salary. It will be great if the boilerplate code can be generated automatically every time we want to traverse the tree structure.

Motivated by this problem of generating generic code for tree structure traversals, more specifically, queries and transformations in object algebras, we designed an object algebra framework with great features. We introduce monoids in queries and generic visitable interfaces in transformations, and write generic query and transformation which can be easily inherited by real cases of queries and transformations. Furthermore, 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 9. Transformation code will be like Figure

3   Queries

How to write generic queries by hand
    Introduce Monoid

```
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;
    }
}
```

Free Variables Monoid
Generic Query by hand with Monoid
Free Vars Query

3

```
public class SalaryQuerySybAlg implements SybAlg<Float,Float,Float,Float,
    Float,Float> {
 public Float C(List<Float> depts){
  Float r = 0f;
  for (Float f: depts) r += f;
  return r;
 }
 public Float D(String name, Float manager, List<Float> subUnits){
  Float r = manager;
  for (Float f: subUnits) r += f;
  return r;
 }
 public Float PU(Float employee){
  return employee;
 }
 public Float DU(Float dept){
  return dept;
 }
 public Float E(Float p, Float s){
  return p + s;
 }
 public Float P(String name, String address){
  return 0f;
 }
 public Float S(float salary){
  return salary;
 }
}
```

Fig. 5. Company Structure represented by Object Algebra Interface

4 Transformations

Before writing generic transformation interfaces, we first introduce *generic visitable interface* as Figure 14. 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 15. Note that the returned *Generic Visitable Interface* will contain all the information for the tree structure.

Now to create a specific transformation, e.g., substitute one variable with another name, it can be easily implemented by inheriting from the generic

```
public interface G_Company {
 <Company,Dept,SubUnit,Employee,Person,Salary> Company accept(SybAlg<
     Company,Dept,SubUnit,Employee,Person,Salary> alg);
}
```

Fig. 6. Generic Visitable Class of Company

```
public interface G_Salary {
  <Company,Dept,SubUnit,Employee,Person,Salary> Salary accept(SybAlg<
      Company,Dept,SubUnit,Employee,Person,Salary> alg);
}
```

Fig. 7. Generic Visitable Class of Salary

transform interface with the implementation of interesting cases. Here only the method G_Exp Var(String ss) needs to be overridden as Figure 16.

5   Object Algebras Framework

Writing generic queries and transformations with monoids and generic visitable interfaces 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. 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/
├── generic/
│   └── G_ExpAlg_Exp
├── query/
│   └── ExpAlgQuery
└── transform/
    └── ExpAlgTransform
```

Furthermore, the monoid interface is also included in the Object Algebras Framework

Hence when programming with query, one only needs to focus on the interesting case, for instance, the return free variable names, and the specific monoid needed, which in Free Variables case will be a String List monoid. While programming with transformation will be implementing interesting cases by inheriting from the AlgNameTransform interface with *generic visitable interfaces*.

5

6 Other Features

One More section

7 Case Study

Case study section

8 Related Work

Finally related work.

9 Conclusion

And conclusion.

Acknowledgements We should thank someone!

References

1. ralf Lammel, S.P.J.: Scrap your boilerplate: A practical design pattern for generic programming. In: TLDI'03 (2003)

```java
public class IncreaseSalarySybAlg implements SybAlg<G_Company, G_Dept,
    G_SubUnit, G_Employee, G_Person, G_Salary>{
  public G_Company C(List<G_Dept> p0) {
    return new G_Company() {
      public <Company,Dept,SubUnit,Employee,Person,Salary> Company accept(
          SybAlg<Company,Dept,SubUnit,Employee,Person,Salary> alg) {
        List<Dept> gp0 = new ArrayList<Dept>();
        for (G_Dept s: p0) {
          gp0.add(s.accept(alg));
        }
        return alg.C(gp0);
      }
    };
  }
  public G_Dept D(java.lang.String p0, G_Employee p1, List<G_SubUnit> p2) {
    return new G_Dept() {
      public <Company,Dept,SubUnit,Employee,Person,Salary> Dept accept(
          SybAlg<Company,Dept,SubUnit,Employee,Person,Salary> alg) {
        List<SubUnit> gp2 = new java.util.ArrayList<SubUnit>();
        for (G_SubUnit s: p2) {
          gp2.add(s.accept(alg));
        }
        return alg.D(p0, p1.accept(alg), gp2);
      }
    };
  }
  public G_SubUnit DU(G_Dept p0) {
    return new G_SubUnit() {
      public <Company,Dept,SubUnit,Employee,Person,Salary> SubUnit accept(
          SybAlg<Company,Dept,SubUnit,Employee,Person,Salary> alg) {
        return alg.DU(p0.accept(alg));
      }
    };
  }
  public G_Employee E(G_Person p0, G_Salary p1) {
    return new G_Employee() {
      public <Company,Dept,SubUnit,Employee,Person,Salary> Employee accept(
          SybAlg<Company,Dept,SubUnit,Employee,Person,Salary> alg) {
        return alg.E(p0.accept(alg), p1.accept(alg));
      }
    };
  }
  public G_Person P(java.lang.String p0, java.lang.String p1) {
    return new G_Person() {
      public <Company,Dept,SubUnit,Employee,Person,Salary> Person accept(
          SybAlg<Company,Dept,SubUnit,Employee,Person,Salary> alg) {
        return alg.P(p0, p1);
      }
    };
  }
  public G_SubUnit PU(G_Employee p0) {
    return new G_SubUnit() {
      public <Company,Dept,SubUnit,Employee,Person,Salary> SubUnit accept(
          SybAlg<Company,Dept,SubUnit,Employee,Person,Salary> alg) {
        return alg.PU(p0.accept(alg));
      }
    };
  }
  public G_Salary S(float p0) {
    return new G_Salary() {
      public <Company,Dept,SubUnit,Employee,Person,Salary> Salary accept(
          SybAlg<Company,Dept,SubUnit,Employee,Person,Salary> alg) {
        return alg.S(1.1f*p0);
      }
    };
  }
}
```

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

Fig. 9. Query Salary Class with Object Algebra Framework

```
public interface SybIncSalary extends SybAlgTransform {
  default G_SybAlg_Salary S(float p0) {
    return new G_SybAlg_Salary() {
      public <Company, Dept, SubUnit, Employee, Person, Salary> Salary
          accept(SybAlg<Company, Dept, SubUnit, Employee, Person, Salary>
          alg) {
        return alg.S(1.1f * p0);
      }
    };
  }
}
```

Fig. 10. Increase Salary Class with Object Algebra Framework

```
public class FreeVarsMonoid implements Monoid<String[]>{
  @Override
  public String[] empty() {
    return new String[]{};
  }
  @Override
  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;
  }
}
```

Fig. 11. Free Variables Monoid

```java
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. 12. Generic Query by hand with Monoid

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

Fig. 13. Free Vars Query

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

Fig. 14. Generic Visitable Interface

9

```java
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. 15. Generic Transformation by hand with acceptor interface

```java
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. 16. Substitute Variables Transformation