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. Traversing complex data structures typically requires large amounts of tedious boilerplate code. For some operations most of the code simply walks the structure, and only a small portion of the code implements the functionality that motivated the traversal in the first place. This paper present a type-safe Java framework called **Shy** that removes much of this boilerplate code. In **Shy** *object algebras* are used to describe complex data structures. Using Java annotations generic boilerplate code is generated for various types of traversals, including queries and transformations. Then programmers can inherit the generic traversal code to focus only on writing the interesting parts of the traversals. Consequentely, the amount of code that programmers need to write is significantly smaller, and traversals using the **Shy** framework are also much more *structure shy*. That is, since traversals have less type-specific code they become much more adaptive to future changes in the data structure. To prove the effectiveness of the approach, we employeed **Shy** on the implementation of a domain-specific questionaire language. Our results show that for a large number of traversals there was a significant reduction in the amount of user-defined code. BRUNO: Say something more about extensibility and type-safety!

1 Introduction

This is the Introduction.

2 Overview

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

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, Employee, SubUnit and Person. A Company comprises a list of Departments. Each Department is managed by an Employee as the manager and contains a

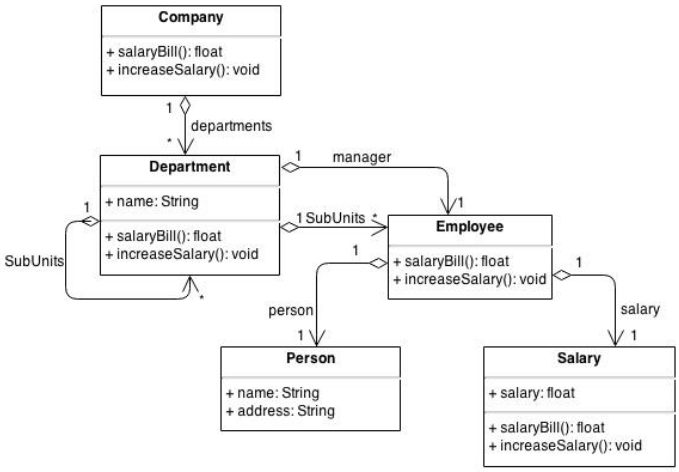

Fig. 1. Company Structure

```
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

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.

```
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

```
@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

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 can implement the Company interface with specific operation for each component.

```
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 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 class FloatQuery implements SybAlgQuery<Float> {
  public Monoid<Float> m() {return new FloatMonoid();}
  public Float S(float p0) {return p0;}
}
```

Fig. 5. Query Salary Class with Object Algebra Framework

```
  }
  ...
  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 we can design some generic classes for queries and transformations. Hence specific algebras can be generated by implementing interesting cases of generic queries and transformations. Moreover, it will be even better if the boilerplate code can be generated automatically so we can focus our attention on the interesting cases.

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, the generic query and transformation classes 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

```
public class IncSalary implements SybAlgTransform<Float, Float, Float,
    Float, Float, Float> {
 private SybAlg<Float, Float, Float, Float, Float, Float> alg;
 public SybAlg<Float,Float,Float,Float,Float,Float> sybAlg() {return alg;}
 public IncSalary(SybAlg<Float, Float, Float, Float, Float, Float> alg) {
     this.alg=alg;}
 public Float S(float salary) {return alg.S(1.1f * salary);}
}
```

Fig. 6. Increase Salary Class with Object Algebra Framework

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:

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*.

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() {
```

```
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;
 }
 @Override
 public Exp Add(Exp p0,Exp p1) {
  Exp res = m.empty();
  res = m.join(res, p0);
  res = m.join(res, p1);
  return res;
 }
 @Override
 public Exp Lit(int p0) {
  Exp res = m.empty();
  return res;
 }
 @Override
 public Exp Var(String p0) {
  Exp res = m.empty();
  return res;
 }
}
```

Fig. 7. Generic Query by hand with Monoid

```
    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 FreeVarsExpAlg extends QueryExpAlg<String[]> {
  public FreeVarsExpAlg(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.

```
public interface StatAlg<Exp, Stat> {
 Stat Seq(Stat s1, Stat s2);
 Stat Assign(String x, Exp e);
}
```

Fig. 8. Statement Algebra Interface

```
public interface DepGraph extends ExpAlg<Set<String>>, StatAlg<Set<String>,
     Set<Pair<String,String>>> {
 default Set<String> Var(String p0) {return Collections.singleton(p0);}
 default Set<String> Lit(int i){return Collections.emptySet();}
 default Set<String> Add(Set<String> e1, Set<String> e2){
  Set<String> deps = new HashSet<>(e1);
  deps.addAll(e2);
  return deps;}
 default Set<Pair<String, String>> Assign(String p0,Set<String> p1) {
  Set<Pair<String,String>> deps = new HashSet<>();
  for (String x: p1) {deps.add(new Pair<>(p0, x));}
  return deps;}
 default Set<Pair<String, String>> Seq(Set<Pair<String, String>> s1, Set<
     Pair<String, String>> s2){
  Set<Pair<String, String>> deps = new HashSet<>(s1);
  deps.addAll(s2);
  return deps;}
}
```

Fig. 9. Dependency Graph

4 Generalized Queries

The previous section discusses simple queries of merging the same type. However, queries can be with different types when various type parameters are passed to the *Object Algebra Interface*. Such generalized version of queries are applicable in more cases and the queries in the previous section is a special case of it.

4.1 Dependency Graph: Query different types

A simple example of generalized queries could be to construct the dependency graph of a program. Let us first extend our simple ExpAlg to a more generalized language StatAlg by adding Statements as in Figure 8.

Now think about constructing the dependency graph from a statement. For $Assign(String x, Exp e)$ method, the variable x will depend on all variables appear in the Expression x. As for $Seq(Stats1, Stats2)$, it is nothing but merge the dependency list appear at both statement dependency lists. A simple implementation of constructing a dependency graph with return type $Set < Pair < String, String >>$ is shown in Figure 9.

Similar to what we have discussed in 3, the traversal code contains too much boilerplates and it is natural to simplify this kind of traversal code in some way.

4.2 Generalized Queries with Monoids

The generalized queries such as constructing the dependency graph as discussed in 4.1 share a lot of similarities. Methods like $Add(Expe1, Expe2)$ and $Seq(Stats1, Stats2)$ can be easily implemented with the help of Monoids, but since generalized queries contain different type arguments, different monoids shall be specified to merge elements with corresponding types.

```
public interface G_StatAlgQuery<A0, A1> extends StatAlg<A0, A1> {
 Monoid<A0> mExp();
 Monoid<A1> mStat();
 default A1 Assign(java.lang.String p0, A0 p1) {
  A1 res = mStat().empty();
  return res;
 }
 default A1 Seq(A1 p0, A1 p1) {
  A1 res = mStat().empty();
  res = mStat().join(res, p0);
  res = mStat().join(res, p1);
  return res;
 }
}
public interface G_ExpAlgQuery<A0> extends ExpAlg<A0> {
 Monoid<A0> mExp();
 default A0 Add(A0 p0, A0 p1) {
  A0 res = mExp().empty();
  res = mExp().join(res, p0);
  res = mExp().join(res, p1);
  return res;
 }
 default A0 Lit(int p0) {
  A0 res = mExp().empty();
  return res;
 }
 default A0 Var(java.lang.String p0) {
  A0 res = mExp().empty();
  return res;
 }
}
```

As shown by the above code, we introduced two monoids, mExp and mStat, to help query the desired dependency graph.

4.3 Dependency Graph with Generalized Query Algebra

Now that we have the Generalized queries with various monoids, it is time to focus on the interesting cases for constructing dependency graph and let the generalized query interface worry about the boilerplates.

Figure 10 shows the code of constructing dependency graph with generalized queries. Similar to simple queries, to specify the desired return type of dependency graph, the developer only needs to clarify the monoids for each type argument.

```
public interface DepGraph extends G_ExpAlgQuery<Set<String>>,
    G_StatAlgQuery<Set<String>, Set<Pair<String,String>>> {
 default Set<String> Var(String p0) {return Collections.singleton(p0);}
 default Set<Pair<String, String>> Assign(String p0,Set<String> p1) {
  Set<Pair<String,String>> deps = new HashSet<>();
  for (String x: p1) {
   deps.add(new Pair<>(p0, x));
  }
  return deps;}
}
```

Fig. 10. Dependency Graph with Generalized Query Algebra

```
class StringPairDepGraph implements DepGraph {
 public Monoid<Set<String>> mExp() {return new SetMonoid<>();}
 public Monoid<Set<Pair<String, String>>> mStat() {
  return new SetMonoid<>();
 }
}
```

Note that the generalized query interface can be implemented with different ways to achieve different desired functionalities and monoids will help specify various query return types.

5 Transformations

Besides the collection of information, one may also want to modify the information under some circumstances, in which case transformation is inspired. Transformation is another essential type of operations handling a data structure. Whereas a programmer unavoidably should also write a bunch of boilerplate code traversing the structure to do some modification.

5.1 SubstVars: a simple tranformation algebra

An example of the transformation algebra, based on the interface ExpAlg is called *substVars*. We anticipate to substitute a given expression for a specific variable. To address this issue, a programmer who has knowledge of object algebras may usually write some traversal code like:

```
public class SubstVarsExpAlg implements ExpAlg<String> {
 String v, e;
 SubstVarsExpAlg(String v, String e) {
  this.v = v;
  this.e = e;
 }
 public String Var(String s) {
  return v.equals(s) ? e : s;
 }
```

```
  public String Lit(int x) {
    return "" + x;
  }
  public String Add(String e1, String e2) {
    return "( " + e1 + " + " + e2 + " )";
  }
}
```

Here v is the name of the specified variable, and e is the substitution. Moreover, this piece of code attaches the transformation to a query of type String, which works as a pretty printer. Nevertheless in that case, transformations are dependent on query algebras, which results in the loss of modularity. Furthermore, with a different transformation algebra, the user still has to write similar traversal code correspondingly, which is very boring.

5.2 The identity approach

In the section of query algebras, we introduce monoids to implement the generic traversal of an object algebra interface. And similarly, the technique to achieve the generic traversal on transformations is called the *identity approach*.

```
public class TransformExpAlg<Exp> implements ExpAlg<Exp> {
  ExpAlg<Exp> alg;
  public TransformExpAlg(ExpAlg<Exp> alg) {this.alg = alg;}
  public Exp Var(String s) {
    return alg.Var(s);
  }
  public Exp Lit(int x) {
    return alg.Lit(x);
  }
  public Exp Add(Exp e1, Exp e2) {
    return alg.Add(e1, e2);
  }
}
```

Here is the generic transformation code of ExpAlg with our identity approach. This class takes an algebra as the incoming argument, and works exactly in the same way as the algebra. In this way transformations become independent of queries, holding the modularity as expected. And though this class is actually doing nothing, a user can simply override some of the methods and get a certain transformation algebra.

Another important characteristic is that we can apply several transformations to the data structure before a query. This pattern is called the *transformation pipeline*. Since a transformation algebra has the same type as the argument its constructor takes, a programmer can define a number of transformation algebras, and make them nested. At the meantime, a query algebra is passed to the innermost constructor. In that case, the query is traversed recursively throughout the pipeline and eventually derives a certain composite transformation algebra.

5.3 Solving substVars with generic transformation algebra

Now with the generic transformation, the substVars transformation can be addressed by simply creating a subclass of it, where some methods are overridden.

```
class SubstVarsExpAlg<Exp> extends ExpAlgTransform<Exp> {
 String v;
 Exp e;
 public SubstVarsExpAlg(ExpAlg<Exp> alg, String v, Exp e) {
  super(alg);
  this.v = v;
  this.e = e;
 }
 public Exp Var(String s) {
  return v.equals(s) ? e : alg.Var(s);
 }
}
```

The SubstVarsExpAlg is still a generic class, however, some query algebra like a pretty printer can be passed to the constructor to display the results. Hence at this moment, a programmer doesn't need to write the boilerplate code for traversals. The identity approach and the pipeline of transformations provide users with a generic transformation like a template.

6 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 itself. It will be even better if the boilerplate code for tree structures can be generated automatically. If we pay more attention to our 4 query and transformation interfaces, without much difficulty we can find that the query and transform code structures for all *Object Algebra Interfaces* share much similarity. Therefore we can make this code generation process automatic.

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
```

```
    ├── G_ExpAlgQuery
└── transform/
    ├── ExpAlgTransform
    └── G_ExpAlgTransform
```

Here the automatically generated ExpAlgQuery, G_ExpAlgQuery, ExpAlgTransform and G_ExpAlgTransform are exactly the same code as we discussed in the previous sections. Furthermore, the monoid interface is also included in the *Object Algebra Framework*.

Hence when programming with query and transformations, 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 SubstVars algebra can be realized by overriding the Exp Var(String s) method of ExpAlgTransform interface, which substitutes variable names as specified.

7 Other Features

One More section

8 Case Study

Case study section

9 Related Work

Object Algebras. Oliveira provided an interesting solution to Expression Problem based on Object Algebras. It is lightweight in terms of required language features without sacrificing extensibility. However, when applying Object Algebras in rich tree structures, boilerplates are hard to avoid, hence motivated our work.

Tree structures traversals. Lammel's Scrap your Boilerplate series introduced a practical design pattern for generic programming in tree structures, which inspired our work of scrapping boilerplates in Object Algebras. Bringert introduced useful compositional functions to help construct final results in the paper A Pattern for Almost Compositional Functions. J. Visser introduced vistor combinators for traversal control in visitor pattern in his paper Visitor Pattern and Traversal Control. The paper dealing with Large Bananas proposed a polytypic programming approach for generalized and basic folds. These fold algebras scale up applications involving large systems of mutually recursive datatypes.

10 Conclusion

And conclusion.

Acknowledgements We should thank someone!

References