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 many 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. Consequently, 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

Many applications require complex recursive data structures. Examples abound in language processing tools/libraries for programming languages, domain-specific languages, markup languages like HTML, or data-interchange languages like XML or JSON. In those applications Abstract Syntax Trees (ASTs) are the key data structure needed to model the various constructs of the languages. Such ASTs have various different types of nodes, which can range from a few dozen to several hundred nodes (for example in the AST of languages like Java).

Static types are helpful to deal with such complex structures. With static types it is easy to distinghish between different kinds of nodes. Furthermore the distinctions are helpful to ensure that traversals over these structures have an appropriate piece of code that deals with each different type of node. This can prevent a large class of run-time errors that would not otherwise be detected.

Unfortunately, for many traversals, the number of nodes and the enforced type distinctions between nodes can lead to so-called boilerplate code []. In this

context, boilerplate code is code that is similar for most different types of nodes and which essentially walks the structure. Traversals where such boilerplate code dominates are called *structure shy* []BRUNO: Who invented the term structure-shy? Alcino Cunha?. In structure shy operations only for a small portion of nodes the code for the core functionality that motivated the traversal in the first place will be different. A typical example would be computing the free variables of an expression for some programming language. In this case, the interesting code would be done in the nodes representing the binding constructs. In all other nodes, the code would just deal with walking the structure. In languages with dozens or hundreds of nodes, having to explictly write cases for each node is a source of significant complexity, since it requires a lot of effort and it is error-prone.

It is possible to avoid boilerplate code in languages that support some kind of reflection [], or using some meta-programming techniques []. With such approaches a programmer only needs to define the code to deal with the nodes that are not structure shy. This has important benefits. Firstly the user has to write much less code, also removing the possibility of errors in the code walking the structure. Secondly the code becomes much more adaptive to changes: if structural changes occur only in the structure shy cases, then the user-defined code remains unchanged.BRUNO: Mention some concrete examples here, or at least cite them. However such approaches are usually done at the cost of type-safety. For example, the use of mechanisms such as Java's dynamic reflection comes at the cost of allowing some run-time errors to occur. Moreover dynamic reflection usually incurs on significant performance penalties. Therefore an important challenge is how to make type-safety and structure shyness co-exist.BRUNO: Talk about SyB.

This paper presents a Java framework called **Shy** that allows users to define *type-safe structure-shy operations*. **Shy** uses *object algebras* to describe complex data structures. Object algebras are a recently introduced technique, which has been shown to have significant advantages for software extensibility. In **Shy** object algebras are combined with java annotations to generate generic boilerplate code for various types of traversals. These traversals include different types of queries and transformationsBRUNO: more detail?. Programmers that want to implement structure shy traversals can inherit the generic traversal code, and focus only on writing the interesting parts of the traversals. Consequently, the amount of code that programmers need to write is significantly smaller, and traversals written in **Shy** are:

- **Adaptive and structure shy: Shy** traversals can omit boilerplate code, allowing these traversals to be more adaptive to future changes in the data structure.
- **Simple and general: Shy** traversals work for any structure that can be expressed as a (multi-sorted) object algebra. This includes complex OO hierarquies or ASTs for large languages.BRUNO: Why simple?
- **Implemented in plain Java: Shy** traversals do not require a new tool or language. The approach is library based and uses only Java annotations.

- **Type-safely reusable:** With **Shy** traversals all reuse is type-safe. No runtime casts are needed for generic traversal code or for user-defined traversal code.
- **Extensible: Shy** traversals inherit the type-safe extensibility from object algebras. Therefore it is possible to reuse traversal code in structures that are extended with additional constructors.

To prove the effectiveness of the approach, we employeed **Shy** on the implementation of the domain-specific questionaire language QL []. Our results show that for a large number of traversals there was a significant reduction in the amount of user-defined code.

Moreover our approach should apply to any OO language with support for generics and annotations, including Scala, or .Net languages.

In summary, the contributions of this paper are:

- **Design patterns for generic traversals.** We provide a set of design patterns for various types of traversals using object algebras. These include: *queries*, *transformations*, *generalized queries* and *generalized transformations*.
- **Shy: a Java framework for eliminating boilerplate code.** We have implemented a Java framework that can be used to describe complex structures using object algebras; and to eliminate boilerplate code. The framework uses Java annotations to automatically generate generic traversals, and is publicly available [1].
- **Case study and empirical evaluation.** We evaluate the approach using a case study based on a domain-specific language for questionaires. The results of our case study show significant savings in terms of user-defined traversal code.

2 Overview

BRUNO: Summary of the Section must come first. Look at the "Extensibility for the Masses" to see how sections are written. In this section, we start by considering a practical problem of representing tree structure in simple object-oriented approach. It turns out that this approach of coding is lengthy and lacks extensibility when the tree structure is big in size. We then applies object algebras to help solving extensibility problem, but the code is still full to tedious delegation code in the tree structure nodes. Thus we introduce generic queries and transformations to make traversal code reusable and modular. Finally we present our framework **Shy** which automatically generate generic queries, generalized generic queries, transformations and contextual aware transformations based on Object Algebra Interfaces with Java annotation.

[1] `www.???.???`

2.1 Object Oriented Solution

We start by considering the company structure introduced in Fig. 1.BRUNO: More text needed. Where does the company example come from? add a reference to it. This example is borrowed from [3], where they use the example to address the problem of boilerplate code when programming with rich tree structures in Haskell.

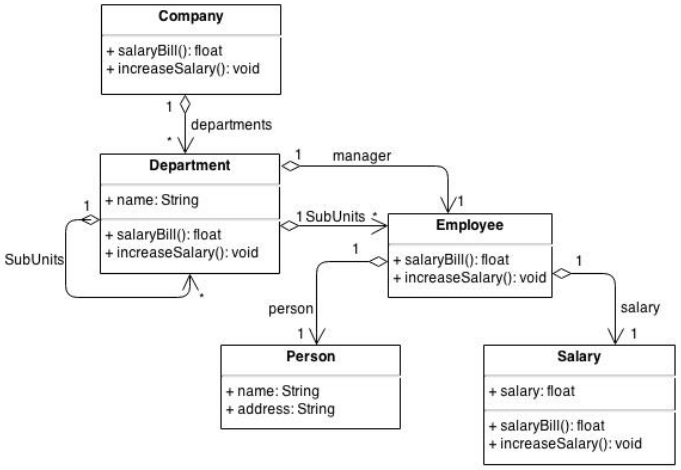

Fig. 1. Company Structure

A very natural Object-Oriented way to model the company structure is as illustrated in Figure 2. Similar code can be applied to `Department`BRUNO: references to code identifiers should be printed with code font as in `Department`.Change all such references throughout the paper., `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.

BRUNO: No need for 2 figures. Just put the code for the two classes in a single figure.

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%. A very natural solution, as illustrated in Fig. 2, is to add methods `Float salaryBill()` and `void increaseSalary()` in all classes from the bottom `class Salary` to the top `class Company`, except some sibling classes like `class Person` which has nothing to do with salary information. Thus query salary bill of the whole company can be implemented as return the salary in `class Salary` and delegating the method `Float salaryBill()` to the child leaves in other upper level classes in the company tree structure, while increase salary of the whole company

```
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();
  }
}
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. 2. Company Class in OOP style

by 10% can be implemented by updating the salary information in `class Salary` and delegating the method `void increaseSalary()` to the child leaves in other upper level nodes. BRUNO: Some more text needed: why do we need to have the methods in most classes, but not in `Person`?

However, this simple object-oriented solution is lack of extensibility and with large amount of boilerplate code when traversing the tree structures.

1. **Lack of extensibility** BRUNO: explain 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 another layer of nodes between classes, such as adding a class Team in our Company example, requires modifying its parent classes and child classes, which violates the no modification rule and are prone to errors. Adding a new operation such as pretty printing of the company structure requires adding a lot of similar methods on the existing code and also violates the no modification rule.
2. **Boilerplate code** BRUNO: explain Another issue of the above solution is that we usually need to write a large amount of boilerplate code to implement very simple task. In our example, we implemented `void increaseSalary()` and `Float querySalary()` in almost all classes, while only `class Salary` does some interesting work, other classes simply delegating the task to its child nodes. This problem become more severe when we have bigger tree structures. The useful code can become very little while most of the code is doing tedious work.

BRUNO: The following text needs to be stronger (use the template above) to better emphasize the two problems that arize here.

```
@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. 3. Company Structure represented by Object Algebra Interface

2.2 Modeling Company Structure with Object Algebras

To tackle with the problem of extensibility, Object Algebras is a good solution. BRUNO: text missing here. We need to introduce object algebras first; explain what they are; cite them; and then talk about the solution. Oliveira proposed the design pattern *Object Algebras*[4] to solve the famous Expression Problem. Object Algebras are classes that implement Object Algebra Interfaces where the type parameters represents the algebra classes. With this extra layer of generic type parameters, Object Algebras is extensible on both data variants and operations.

Fig. 3 shows the approach to model the Company structure as an object algebra interface. Different operations can be realized by extending object algebras inheriting from the object algebra interface. To implement query bill operation for the whole Company structure, we can pass in a `Float` class to represent salary and implement the Company interface with specific operation for each component. In our case, the `QuerySalarySybAlg` returns salary information in method `Float S(float salary)` and does the delegation work in most other methods related to salary information.

```
public class QuerySalarySybAlg 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;
  }
}
```

IncreaseSalary will modify an old algebra and return a new algebra based on specific implementations. More specifically, the old algebra is set as a field in `IncreaseSalarySybAlg`, method `Salary S(float salary)` increases the salary by 10%, and other methods simply delegates to the same methods in the old algebra.

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

BRUNO: The code is too specific. It should be something like: `public class IncreaseSalarySybAlg<Company, Dept, SubUnit, Employee, Person, Salary> implements SybAlg<Company, Dept, SubUnit, Employee, Person, Salary>`

BRUNO: code needs to be better explained. What are we using the `alg` for?

When programming with Object Algebras, the code can be very generic based on the object algebra interfaces. `alg` can be used as an abstract factory to construct algebras, while the algebras can be instantiated by specific object algebras later by passing in type parameters.

However, although we solved the problem of extensibility with object algebras, the traversal code is still lengthy and we are still writing tedious traversing code of tree structures. 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 better 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, generalized generic queries, transformations and contextual aware transformations with Object Algebras, which can be easily inherited by real cases of queries and transformations. Furthermore, we designed an object algebra framework **Shy** . 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 Fig. 4. Here when coding with the framework **Shy** , users need not worry about any boilerplate delegation code of the tree structure, but just simply rewrite the interesting case which actually return the salary information.

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

Fig. 4. Query Salary Class with Object Algebra Framework

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

Fig. 5. Increase Salary Class with Object Algebra Framework

Transformation code will be like Figure 5. Similar to the query example, all the boilerplate part has been handled by the framework and the developer only needs to pass in the original company algebra and override the interesting case `Float S(float salary)`, then a new company algebra will be returned after transformation.

BRUNO: code is too specific. See comment about previous increase salary example BRUNO: Jason, generally speaking your explanations of the code are to brief: you don't actually explain the code. You need to emphasize the relevant parts of the code, as well as the parts that are non-obvious. Here for example you want to emphasize that we only need to write the salary method.

3 Queries

As a specific type of object algebras, queries allow users to define new operations handling a user-defined data structureBRUNO: This definition of queries is just too broad. A query is an operation that traverses a structure and computes some aggregate value. Please look at papers like Syb to see how they describe queries and transformations. 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 numeric literal, a variable or the addition of two expressions. 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);
}
```

Fig. 6. Object algebra interface: ExpAlg

Based on the interface above, a query might be raised on collecting all the names of free variables defined in an expression. More precisely, a list of strings would be used to store the names of those variables. In that case, a `Var(s)` would simply return a singleton list of `s`, and a `Lit(i)` corresponds to an empty list, whereas two lists would be joined 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 Fig. 7.

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.

```
public interface FreeVarsExpAlg extends ExpAlg<List<String>> {
  default List<String> Var(String s) {
    return Collections.singletonList(s);
  }
  default List<String> Lit(int i) {
    return Collections.emptyList();
  }
  default List<String> Add(List<String> e1, List<String> e2) {
    List<String> res = new ArrayList<String>(e1);
    res.addAll(e2);
    return res;
  }
}
```

Fig. 7. A normal algebra-based approach for freeVars

```
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;
   }
}
```

Fig. 8. A generalized monoid interface

```
public interface ExpAlgQuery<Exp> extends ExpAlg<Exp> {
 Monoid<Exp> m();
 default Exp Var(String s) {
  Exp res = m().empty();
  return res;
 }
 default Exp Lit(int i) {
  Exp res = m().empty();
  return res;
 }
 default Exp Add(Exp e1, Exp e2) {
  Exp res = m().empty();
  res = m().join(res, e1);
  res = m().join(res, e2);
  return res;
 }
}
```

Fig. 9. Generic query by hand with monoid

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.

The interface of a monoid is defined in Fig. 8. Intuitively, the `join()` method implies how we combine 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. See Fig. 9.

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, at first we can simply override the method `m()`, which provides an instance of `Monoid`, in the return statement. As the next step, we only need to override a few other methods to meet the requirements. This is the progress, once we have 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 class FreeVarsMonoid implements Monoid<List<String>> {
  public List<String> empty() {
    return Collections.emptyList();
  }
  public List<String> join(List<String> e1, List<String> e2) {
    List<String> res = new ArrayList<String>(e1);
    res.addAll(e2);
    return res;
  }
}
```

Fig. 10. A monoid instance defined for freeVars

```
public interface FreeVarsExpAlg extends ExpAlgQuery<List<String>> {
  default Monoid<List<String>> m() {
    return new FreeVarsMonoid();
  }
  default List<String> Var(String s) {
    return Collections.singletonList(s);
  }
}
```

Fig. 11. The query interface for freeVars

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 sub-interface of `ExpAlgQuery`, the generic algebra, with generic type to be `List<String>`. To use the generic traversal code, a monoid is defined in Fig. 10.

BRUNO: Is an array the best structure to use here? Wouldn't a vector or list be better? The simpler the code is, the better.

But the result for an expression can only be a null list based on the monoid. Thus in the freeVars query, furthermore, we expect the variables to store their names into a list, and by using the monoid, freeVars can be implemented. See Fig. 11.

When the interface `FreeVarsExpAlg` is used, an object of the `FreeVarsMonoid` is then created. As we can see, it is needless for a user to write an exclusive traversal fully for 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 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.

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

Fig. 12. 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. 13. Dependency Graph

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 Fig. 12. Two more statements, sequence and assign operation, are added to our statement language.

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 `e`. As for `Seq(Stat s1, Stat s2)`, 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 Fig. 13. Here `Var(String p0)` is the source where variable are created, and `Assign(String p0, Set<String> p1)` describes that all variable in p1 are dependent on variable p0. Other methods only congregate same type of variables to larger collections.

Similar to what we have discussed in Section 3, the traversal code contains boilerplates and it is natural to simplify this kind of traversal code in a similar 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(Exp e1, Exp e2)` and `Seq(Stat s1, Stat s2)` 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.

JASON: The example implementing two interfaces seperately is not good as `mExp()` in G_StatAlgQuery is not actually used.

```
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;
 }
}
```

JASON: I should explain more of the code, specifically, how various monoids work together and build large tree structures. What is the algorithm of defining default implementation in constructor methods With various typed arguments.

As shown above, if we introduce two monoids, `mExp` and `mStat`, it is simple to query and construct the desired dependency graph with monoids specific to the query types.

```
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. 14. Dependency Graph with Generalized Query Algebra

4.3 Dependency Graph with Generalized Query Algebra

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

Fig. 14 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.

```
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, by which transformation is inspired. Transformation is another essential type of object algebras with respect to query. Whereas a programmer unavoidably should also write a bunch of boilerplate code traversing the structure make some changes.

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 Fig. 15.

```
public interface SubstVarsExpAlg<Exp> extends ExpAlg<Exp> {
 ExpAlg<Exp> expAlg();
 String getVar();
 Exp getExp();
 default Exp Var(String s) {
  return s.equals(getVar()) ? getExp() : expAlg().Var(s);
 }
 default Exp Lit(int i) {
  return expAlg().Lit(i);
 }
 default Exp Add(Exp e1, Exp e2) {
  return expAlg().Add(e1, e2);
 }
}
```

Fig. 15. A normal algebra-based approach for substVars

In Fig. 15, `getVar()` is the name of the specified variable, and `getExp()` is the substitution. `expAlg()` is an instance of `ExpAlg` on which the transformation is based. Here comes the same problem. The boilerplate code consists of a particular traversal, with implementing all methods in the algebra interface to fulfill the requirements. It implies redundant work for a different transformation.

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 interface ExpAlgTransform<Exp> extends ExpAlg<Exp> {
 ExpAlg<Exp> expAlg();
 default Exp Var(String s) {
  return expAlg().Var(s);
 }
 default Exp Lit(int i) {
  return expAlg().Lit(i);
 }
 default Exp Add(Exp e1, Exp e2) {
  return expAlg().Add(e1, e2);
 }
}
```

Fig. 16. Generic transformation by hand with identity approach

Fig. 16 shows 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 the transformation algebra `ExpAlgTransform<Exp>` is a sub-interface of `ExpAlg<Exp>`, it creates any instance with the same type as the `expAlg()` inside. Therefore a programmer can define a number of transformation algebras, and compose them in sequence, like a nested structure. At the meantime, a query algebra is passed to the innermost `expAlg()`. 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 our generic transformation, one solution to substVars is available in Fig. 17, where the method `Var()` from its super-interface `ExpAlgTransform` is overridden.

```
public interface SubstVarsExpAlg<Exp> extends ExpAlgTransform<Exp> {
 String getVar();
 Exp getExp();
 default Exp Var(String s) {
  return s.equals(getVar()) ? getExp() : expAlg().Var(s);
 }
}
```

Fig. 17. The transformation interface for substVars

The `SubstVarsExpAlg` is still generic, yet something like a pretty printer can be passed to the `expAlg()` inside as a query, to display the results. Hence at this moment, a programmer doesn't need to write 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, generalized generic queries, generic transformations and contextual generic 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. Ideally the boilerplate code for tree structure traversal should be generated automatically so the developers can always focus on methods with interesting things. 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 a framework **Shy** , which utilizes *Java Annotation* to generate query and transformation interfaces based on the *Object Algebra Interface*. For the below code:

```
@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. Developers can implement interesting methods by inheriting from these generic classes without worrying about the traversing work.

Furthermore, the monoid interface is also included in the *Object Algebra Framework*. Hence users can design monoids inheriting from our generic monoid.

With our framework **Shy** , 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 `class ExpAlgQuery` to return variable name, and provide the specific monoid needed, which in this case will be a `StringListMonoid`. 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

To the best of our knowledge, few work has been done addressing writing generic queries and transformations with object algebras, while we should mention some work on object algebras and large tree structures traversal control, which inspired and formed the basis of our work.

Object Algebras. Oliveira proposed Object Algebras as a solution to Expression Problem. Object algebras applies well in mainstream object oriented languages like Java. It is a lightweight solution in terms of language features required. Oliveira also worked on using intersection type and type-constructor polymorphism to make object algebras compositional with feature oriented programming. Different from their prior research, we found that when applying object algebras in rich tree structures, boilerplate code is hard to avoid. Hence our work focuses on reducing the amount of boilerplate code for developers when writing queries and transformations with object algebra tree structures.

Tree structures traversals. In the functional programming community like Haskell, much research has been done on traversal control of large structures. Lammel's Scrap your Boilerplate[3] series introduced a practical design pattern for generic programming in tree structures, which inspired our work of scrapping boilerplates in Object Algebras. Bringert[1] introduced useful compositional functions to help construct final results in Haskell. Lammel[2] proposed a polytypic programming approach for generalized and basic folds. These fold algebras scale up applications involving large systems of mutually recursive datatypes. These works all try to optimize traversal control of large structures in functional programming paradigm, while our work solves a similar problem in Object Algebras, a programming style in Object Oriented Programming paradigm.

Visitor Combinators and Traversal Control. Visser[5] provided some visitor combinators that can express interesting traversal strategies in visitor pattern. We applies some similar idea like identity transformation in simple transformation, but our work targets at traversal control in Object Algebras.

In summary, prior to our work, research has been done on object algebras and composition problem of this programming style. In the functional programming world and with visitor pattern, traversal control in large structures is also explored. Different from these work, we explored techniques helping write generic queries and transformations traversing large tree structures with Object Algebras.

10 Conclusion

And conclusion.

Acknowledgements We should thank someone!

References

1. Bjorn Bringert, A.R.: A pattern for almost compositional functions. Journal of Functional Programming 18, 567–598 (September 2008)
2. Ralf Lammel, Joost Visser, J.K.: Dealing with large bananas (2000)
3. Ralf Lammel, S.P.J.: Scrap your boilerplate: A practical design pattern for generic programming. In: TLDI'03 (2003)
4. Bruno C. d. S. Oliveira, W.R.C.: Extensibility for the masses, practical extensibility with object algebras. In: ECOOP'12 (2004)
5. Visser, J.: Visitor combination and traversal control. In: OOPSLA '01 (2001)