Scrap your Boilerplate with Object Algebras

Name1
Affiliation1
Email1

Name2 Name3
Affiliation2/3
Email2/3

Abstract

Traversing complex *Abstract Syntax Trees* (ASTs) 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 presents a type-safe Java framework called **Shy** that removes much of this boilerplate code. In **Shy** *object algebras* are used to describe complex and extensible AST structures. Using Java annotations **Shy** generates generic boilerplate code for various types of traversals. For a concrete traversal, users of **Shy** can then inherit from the generated code and override only the interesting cases. Consequently, the amount of code that users need to write is significantly smaller. Moreover, traversals using the **Shy** framework are also much more *structure shy*, becoming more adaptive to future changes or extensions to the AST structure. To prove the effectiveness of the approach, we applied **Shy** in the implementation of a domain-specific questionnaire language. Our results show that for a large number of traversals there was a significant reduction in the amount of user-defined code.

1. Introduction

Various language processing tools or libraries for programming languages, domain-specific languages, mark-up languages like HTML, or data-interchange languages like XML or JSON require complex *Abstract Syntax Tree* (AST) structures. In those applications ASTs are the key data structure needed to model the various constructs of these languages. Such ASTs have various different types of nodes, which can range from a few dozen to several hundred kinds of nodes (for example in the ASTs of languages like Java or Cobol).

Static types are helpful to deal with such complex ASTs. Static types formalize the distinction between different kinds of nodes. Furthermore the distinctions are helpful to ensure that traversals over these ASTs 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, when traversing such ASTs, the number of nodes and the enforced type distinctions between nodes can lead to so-called *boilerplate code* [13]: code that is similar for most types of nodes and which essentially just walks the structure. Operations where the "interesting" code is limited to only a small portion of nodes are called *structure shy* [19]. A typical example is computing the free variables of an expression in some programming language. In this case, the interesting code occurs in the nodes representing the binding and variable constructs. In all other cases, the code would just deal with walking the structure. In ASTs with dozens or hundreds of kinds of nodes, having to explicitly write code for each kind of node is both tedious and error-prone.

The boilerplate problem in implementing traversals has received considerable attention in the past. For example, both *Adaptive Object-Oriented Programming* (AOOP) [19] and *Strategic Programming* [2, 29] are aimed partly at solving this problem. Most approaches to AOOP and strategic programming use some meta-programming techniques, such as code generation or reflection. The use of meta-programming offers programmers an easy way to avoid having to write boilerplate code. This has important benefits: users have to write much less code; and the code becomes much more adaptive to changes. However such meta-programming based approaches usually come at the cost of other desirable properties, such as type-safety, extensibility or separate compilation. The functional programming community has also studied the problem before. For instance, the popular "*Scrap your boilerplate*" [13] approach supports type-safety and separate compilation. However most of the techniques used in functional languages cannot be easily ported to mainstream OO languages like Java, and are limited in terms of extensibility.

This paper presents a Java framework called **Shy** that allows users to define type-safe and extensible structure-shy operations. **Shy** uses *Object Algebras* [21] to describe ASTs, and to write operations over ASTs in a style similar to writing folds in functional programming. However unlike standard folds in functional programming, Object Algebras are *extensible*: when new kinds of nodes are introduced in the data type, the operations can be extended as well, without changing existing code.

In **Shy** Object Algebra interfaces are combined with Java annotations to generate generic traversal code automatically. The generated code accounts for four different types of traversals: *queries*; *transformations*; *generalized queries*;

and *contextual transformations*. Each of these four types of traversals is implemented as an Object Algebra. Programmers who want to implement structure-shy traversals can then inherit from one of these four object algebras, and override only the cases that deal with the interesting parts of the traversal. Consequently traversals written in **Shy** are:

- **Small in size:** With **Shy** the amount of code that programmers need to write a structure-shy traversal is significantly smaller. Often traversals in **Shy** are implementable in just a few lines of code, even for complex ASTs with hundreds of different types of nodes.
- **Adaptive and structure shy:** Traversals written with **Shy** can omit boilerplate code, making traversals more adaptive to future changes or extensions to the data type.
- **Type-safe: Shy** traversals are directly written in Java and the Java type-system ensures type-safety. No run-time casts are needed for generic traversal code or for user-defined traversal code.
- **Extensible with separate compilation:** Traversals inherit type-safe extensibility from object algebras. Both traversals and the AST structures are extensible. Thus it is possible to reuse traversal code for ASTs extended with additional node types. Furthermore **Shy** traversals support separate compilation.
- **Implemented in plain Java: Shy** traversals do not require a new tool or language. The approach is library based and only uses Java annotations.

To prove the effectiveness of the approach, we have applied **Shy** in the implementation of QL, a domain-specific language (DSL) for defining questionnaires that has been used before as a benchmark language [7, 10]. Our results (see details in Section 10) show that for a large number of traversals there was a significant reduction in the amount of user-defined code: only 4% to 21% of the AST cases had to be implemented in comparison with code written without **Shy**.

Although **Shy**'s functional programming inspired idioms are new to mainstream Java programmers, its basis on standard Java mechanisms makes **Shy** easy to implement, integrate in existing environments, and use. Moreover, although Java was chosen as the implementation language for **Shy**, our approach should apply to any object-oriented language with support for generics and annotations (for example Scala or C#).

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 *contextual transformations*.
- **The Shy Java framework.** We show that those design patterns can be automatically generated for a given object algebra interface. The **Shy** framework[1] realizes this idea by using Java annotations to automatically generate generic traversals.
- **Case study.** We illustrate the benefits of **Shy** using a case study based on the QL domain-specific language. The results of our case study show significant savings in terms of user-defined traversal code. The case study also shows that **Shy** does not incur significant performance overhead compared to a regular AST-based implementation.

2. Object Algebras

Object Algebras capture a design pattern to solve the Expression Problem [32]. As a result Object Algebras support modular and type-safe extensibility in two dimensions: data variants and operations. Object Algebras can be used conveniently to model recursive data types, such as ASTs. Here we briefly introduce Object Algebras using the example of a simple expression language.

```
interface ExpAlg<E> {
  E Lit(int n);
  E Add(E l, E r);
}
```

The above code shows an *Object Algebra interface* which models `Lit` and `Add` expressions of `ExpAlg`. The interface resembles abstract factory interface, but instead of constructing objects of a concrete type, the result is abstract through the type parameter `E`. A concrete Object Algebra will implement such an interface instantiating the type parameter to construct objects for a particular purpose.

For instance, the class `Eval` which implements evaluation of expression is as follows:

```
class Eval implements ExpAlg<Integer> {
  public Integer Lit(int n) {
    return n;
  }
  public Integer Add(Integer l, Integer r) {
    return l + r;
  }
}
```

Using this algebra to create expression in fact immediately evaluates the expression to its result. To add another operation the interface is simply implemented again. As an example, the class `Print` presented below shows a simple printing operation on expressions. Both examples illustrate extension in the dimension of operations.

```
class Print implements ExpAlg<String> {
  public String Lit(int n) { return "" + n; }
  public String Add(String l, String r) {
    return l + " + " + r;
  }
}
```

[1] **Note to Reviewers:** Due to the anonymous review process we submit a file bundle with the implementation together with the paper submission. If the paper is accepted we will make the code publicly available.

The other dimension of extensibility (adding data variants) is realized by extending the object algebra interface itself. For instance, the interface `MulAlg` defined below extends the original `ExpAlg` with multiplication expressions.

```
interface MulAlg<E> extends ExpAlg<E> {
 E Mul(E l, E r);
}
```

The key feature of Object Algebras is that the existing operations (evaluation and printing) can now be extended to support multiplication without having to change the existing `Eval` and `Print` classes. To realize this for evaluation, some code can be written as follows:

```
class MulEval extends Eval
 implements MulAlg<Integer> {
  public Integer Mul(Integer l, Integer r) {
    return l * r;
  }
}
```

In summary, Object Algebra interfaces describe recursive data types, and implementations of those interfaces represent operations. Extending the interface allows developers to extend the data type with new data variants, and (re)implementing the interface allows developers to define operations. Finally, extension in both dimensions is fully type safe and does not compromise separate compilation.

3. Overview of Shy

This section starts by illustrating the problem of boilerplate code when implementing traversals of complex structures. It then shows how **Shy** addresses the problem using a combination of Object Algebras [21] and Java annotations.

3.1 Traversing Object-Oriented ASTs

We start by introducing the problem boilerplate code by considering a simplified variant of the QL language used in our case study [10], called MiniQl. Just like QL, MiniQL can be used to describe interactive questionnaires. An example is shown in Figure 1. The questionnaire first asks for the user's name and age, and then, if the age is greater than or equal to 18, asks if the user has a driver's license. Because of the conditional construct, the last question only will appear when the user is actually eligible to have driver's license.

MiniQL's abstract syntax contains forms, statements (if-then and question) and expressions (only literals, variables and greater-than-or-equal). A traditional OO implementation is shown on the left Figure 2. A form (class `Form`) has a name and consists of a list of statements. Statements are conditionals (`If`) which contain an expression and a statement body, and questions (`Question`) which have a name, label and type. Expressions are standard, but limited to literals (`Lit`), variables (`Var`) and greater-than-or-equal (`GEq`).

The code in Figure 2 also shows a query over MiniQL structures, namely the collection of used variables. The operation is defined using the method `usedVars`, declared in

```
form DriverLicense {
  name: "What is your name?" string
  age: "What is your age?" integer
  if (age >= 18)
    license: "Have a driver's license?" boolean
}
```

Figure 1. Example QL questionnaire: driver's license

```
@Algebra
public interface QLAlg<E, S, F> {
 F Form(String name, List<S> body);
 S If(E cond, S then);
 S Question(String name,String label,String type);
 E Lit(int n);
 E Var(String x);
 E GEq(E lhs, E rhs);
}
```

Figure 3. Object Algebra interface of the MiniQL abstract syntax

the abstract superclasses `Stmt` and `Exp` (omitted for brevity), and implemented in the concrete statement and expression classes. As can be seen, the only interesting bit of code is the `usedVars` method in class `Var`. All other implementations merely deal with aggregating results of their child nodes, or returning a default empty set.

The boilerplate code exhibited in the `usedVars` query often also applies to transformations. Consider for example a rename transformation which takes a `Form` and returns another form where all variables are renamed. Again, the only interesting case is in the `Var` class, where the actual renaming is applied. All other classes, however, require boilerplate to recreate the structure. The full code in Appendix A.1.1 contains a simple example of such a rename as well.

In addition to the significant amount of boilerplate code, there is another drawback to the traditional OO solution, which is that it does not support extensibility along the dimension of operations. Each new operation requires pervasive change across the AST classes.

3.2 Modeling MiniQL with Object Algebras

The right-hand side of Figure 2 shows the used variables operation implemented in the Object Algebra style. The operation is a class implementing the Object Algebra interface (`QlAlg` shown in Figure 3.

The `UsedVars` class provides an implementation for each of the methods in the object algebra interface, which together define the full used variables operation. Since the result of collecting those variables is `Set<String>`, all the type parameters are set to that type. Most of the method implementations simply traverse the child nodes and accumulate the variable names. That is the case, for example, for `Form`. Again, the only method implementation that does something different is `Var`, which returns the `x` argument.

```
class Form {
 String name; List<Stmt> body;
 Set<String> usedVars() {
  Set<String> vars = new HashSet<>();
  body.forEach(s -> vars.addAll(s.usedVars()));
  return vars;
 }
}

class If extends Stmt {
 Exp cond; Stmt then;
 Set<String> usedVars() {
  Set<String> vars=new HashSet<>(cond.usedVars());
  vars.addAll(then.usedVars());
  return vars;
 }
}

class Question extends Stmt {
 String name, label, type;
 Set<String> usedVars() { return emptySet(); }
}

class Lit extends Exp {
 int n;
 Set<String> usedVars() { return emptySet(); }
}

class Var extends Exp {
 String x;
 Set<String> usedVars() { return singleton(x); }
}

class GEq extends Exp {
 Exp lhs, rhs;
 Set<String> usedVars() {
  Set<String> vars = new HashSet<>(lhs.usedVars());
  vars.addAll(rhs.usedVars());
  return vars;
 }
}
```

```
class UsedVars implements
   QLAlg<Set<String>, Set<String>, Set<String>> {

 Set<String> Form(String n, List<Set<String>> b) {
  Set<String> vars = new HashSet<>();
  b.forEach(s -> vars.addAll(s));
  return vars;
 }

 Set<String> If(Set<String> c, Set<String> t) {
  Set<String> vars = new HashSet<>(c);
  vars.addAll(t);
  return vars;
 }

 Set<String> Question(String n,String l,String t) {
  return Collections.emptySet();
 }

 Set<String> Lit(int x) {
  return Collections.emptySet();
 }

 Set<String> Var(String x) {
  return Collections.singleton(x);
 }

 Set<String> GEq(Set<String> l, Set<String> r) {
  Set<String> vars = new HashSet<>(l);
  vars.addAll(r);
  return vars;
 }
}
```

Figure 2. Implementing the "used variables" operation using traditional ASTs (left) and Object Algebras (right)

Although the implementation still exhibits the same amount of boilerplate code as the traditional AST-based implementation, the Object Algebra style support adding operations without changing existing code. For instance, the renaming operation mentioned above could be realized as follows:

```
class Rename<E, S, F> implements QLAlg<E, S, F> {
 QLAlg<E, S, F> alg;

 F Form(String n, List<S> b) {
  return alg.form(n, b);
 }
 S Question(String n, String l, String t) {
  return alg.question(n + "_", l, t);
 }
 ...
 E Var(String x) { return alg.var(x+ "_"); }
}
```

Each constructor reconstructs a new node using an auxilar MiniQL algebra called `alg`. Almost all the method implementations reconstruct the structure with no changes using the methods in `alg`. For instance, the `Form` method just recreates the form in the algebra `Alg`. The two exceptions are the methods `Question` and `Var`, where the identifiers are suffixed with "_".

Although the Object Algebra encoding of MiniQL solves the problem of extensibility, the traversal code still contains boilerplate code. In both `UsedVars` and `Rename`, the only interesting code is in a small number of cases. Ideally, we would like to write only the code for the interesting cases, and somehow "inherit" the tedious traversal code.

3.3 Shy: An Object Algebra Framework for Traversals

To deal with the boilerplate problem we created **Shy**: a Java object algebras framework, which provides a number of

```
class UsedVars implements QLAlgQuery<Set<String>> {
  public Monoid<Set<String>> m() {
    return new SetMonoid<String>();
  }
  public Set<String> var(String name) {
    return Collections.singleton(name);
  }
}
```

Figure 4. MiniQL used variables, implemented with **Shy**

```
class Rename<E, S, F> extends QLAlgTrans<E, S, F> {
  public Rename(QLAlg<E, S, F> alg) { super(alg); }
  public S question(String n, String l, String t) {
    return qLAlg().Question(n + "_", l, t);
  }
  public E var(String x) {
    return qLAlg().Var(x + "_");
  }
}
```

Figure 5. MiniQL renaming, implemented with **Shy**.

generic traversals at the cost of a single annotation. The key idea in **Shy** is to automatically create highly generic Object Algebras, which encapsulate common types of traversals. In particular **Shy** supports generic *queries* and *transformations*. The two types of traversals are, for instance, sufficient to capture the used variables and renaming operations.

With **Shy**, programmers just need to add the `@Algebra` annotation to the definition of `QLAlg` to get the code for generic queries and transformations. An example of that annotation is already shown in Figure 3. Triggered by the annotation, **Shy** generates base traversal interfaces with Java 8 **`default`** methods which can then overridden to implement specific behavior. For instance, for the MiniQL algebra, **Shy** generates interfaces `QLAlgQuery` and `QLAlgTrans` which can be used to implement `UsedVars` and `Rename` in only a fraction of the code[2].

The **Shy**-based implementation of both operations is shown in Figure **??**. By implementing the `QLAlgQuery` and `QLAlgTrans` interface, only the methods `question` and `var` need to be overridden: all the other methods perform basic accumulation for queries and basic identity reconstruction in the case of transformation. For queries the only extra thing a programmer has to do is to provide an instance of a monoid, which is used to specify how to accumulate the results during the traversal. Similarly, for transformations, the programmer needs to pass an algebra for providing the constructors for creating the result of a transformation.

To use the queries and operations on a questionnaire like the one in Figure 1. we again need a function to create a structure using the generic MiniQL interface:

```
  <E, S, F> F makeQL(QLAlg<E, S, F> alg) {
    return alg.Form("DriverLicense", Arrays.asList(
       alg.Question("name", "Name?", "string"),
       alg.Question("age", "Age?", "integer"),
       alg.If(alg.Geq(alg.Var("age"),
                      alg.Lit(18)),
                alg.Question("license",
                   "License?", "boolean"))));
  }
```

Since both queries and transformations are implementations of the MiniQL interface, they can be passed to the `makeQL` function defined above:

```
println(makeQL(new UsedVars()));
println(makeQL(new Rename<>(new UsedVars())));
```

This code prints out `[age]` and `[age_]`, which are the set of used variables before and after renaming, respectively. Note how the `Rename` transformation transforms the questionnaire into the the `UsedVars` algebra.

The remainder of the paper provides the details and implementation techniques used in **Shy**. Besides basic queries and transformations, **Shy** also supports two generalizations of these types of traversals called *generalized queries* and *contextual transformations*.

4. Queries

This section shows the ideas behind generic queries and how they are implemented in **Shy**. A query is an operation that traverses a structure and computes some aggregate value. The inspiration for queries comes from similar types of traversals used in functional programming libraries, such as "Scrap your Boilerplate" [13].

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

The above code shows a simple type of expressions, represented as the object algebra interface `ExpAlg`. We will use this minimal object algebra interface throughout the rest of the paper to illustrate the various different types of traversals supported by **Shy**. Three different kinds of nodes exist: a numeric literal, a variable or the addition of two expressions. Queries are illustrated by implementing an operation to compute the free variables in an expression.

4.1 Boilerplate Queries

Fig. 6 shows a standard approach for computing free variables using object algebras[3]. A set of strings is used to store the names of the free variables. The `Var` method returns a singleton set of `s`, whereas the `Lit` method returns an empty set. The more interesting case is in the `Add` method, where the two sets are joined into one.

[2] The generated code is available in Appendix A.1.4 and A.1.5. And the interface of `Monoid` in Section 4.2.

[3] Here and in the following we will use interfaces with **`default`**-methods (as introduced in Java 8) to combine queries and transformations using multiple inheritance.

```
interface FreeVars extends ExpAlg<Set<String>> {
 default Set<String> Var(String s) {
  return Collections.singleton(s);
 }
 default Set<String> Lit(int i) {
  return Collections.emptySet();
 }
 default Set<String> Add(Set<String> e1,
    Set<String> e2) {
  return Stream.concat(e1.stream(), e2.stream())
     .collect(Collectors.toSet());
 }
}
```

Figure 6. Free variables as an object algebra.

The typical pattern of a query is to collect some information from some of the nodes of the structure, and to aggregate the information that comes from multiple child nodes. For example, in the case of free variables, the strings from the `Var` nodes are collected, and in the `Add` nodes the information from multiple children is merged into a single set. An important observation about queries is that the code to aggregate information tends to be the same: if we had a subtraction node, the code would be essentially identical to `Add`. Moreover, there are only very few types of nodes that contain relevant information for the query. For nodes that contain information that is not relevant to the query, we simply return the a neutral value (such as the empty set in `Lit`). Nonetheless, a programmer has to write this boring boilerplate code handling the traversals. While for the small structure presented here this may not look too daunting, in a large structure with dozens or even hundreds of constructors such code becomes a significant burden.

4.2 Generic Queries

Clearly a better approach would be to abstract the generic traversal code for queries. This way, when programmers need to implement query operations, they can simply reuse the generic traversal code and focus only on dealing with the nodes that do something interesting.

The code that captures the aggregation and collection of information can be captured by a well-known algebraic structure called a *monoid*. Monoids are commonly used in functional programming for such purposes, but they are perhaps less commonly known in object-oriented programming. The interface of a monoid is defined as follows:

```
interface Monoid<R> {
 R join(R x, R y);
 R empty();
}
```

Intuitively, the `join()` method is used to combine the information from substructures, and the `empty()` is an indicator of "no information". For instance, accumulate results in a set, we could use the following `SetMonoid`:

```
class SetMonoid<X> implements Monoid<Set<X>> {
 public Set<X> empty() {
  return Collections.emptySet();
 }
 public Set<X> join(Set<X> x, Set<X> y) {
  Set<X> tmp = new HashSet<>(x);
  tmp.addAll(y);
  return tmp;
 }
}
```

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

Figure 7. Generic queries using a monoid.

Using the monoid operations alone, it is possible to write a "generic query". Fig. 7 shows how this is achieved. In nodes that contain child nodes, such as `Add`, the information is aggregated using `join`. In nodes that contain other information, such as `Var` and `Lit`, the query returns `empty`. This allows concrete queries to be implemented by overriding methods from multiple, different algebras.

4.3 Free Variables with Generic Queries

The `ExpAlgQuery` interface provides an alternative way to define the free variables operation. Since the result of free variables is a set, the monoid returned by `m()` is an implementation of the `Monoid` interface, where `empty()` corresponds to the empty set, and `join()` is implemented as union. Using this monoid the free variables operation is defined as follows:

```
interface FreeVars extends ExpAlgQuery<Set<String>>
{
 default Monoid<Set<String>> m() {
  return new SetMonoid<String>();
 }
 default Set<String> Var(String s) {
  return Collections.singleton(s);
 }
}
```

There are two important differences to the implementation in Fig. 6. Firstly, the monoid to be used needs to be specified. Secondly only the case for variables needs to be defined: the other cases are inherited from `ExpAlgQuery`.

The traversal code in `ExpAlgQuery` is entirely mechanical and can be automatically generated. This is precisely what **Shy** does. Annotating algebra interfaces, such as `ExpAlg`, with the annotation `@Algebra`, triggers automatic generation of generic query interfaces, such as `ExpAlgQuery`. Fig. 8 shows the general template in **Shy** for an algebra `Alg<`X_1`, ..., `X_n`>`, with constructors `f`$_1$`, ...,` `f`$_m$. Note that interface `Alg`$_Q$ extends `Alg` so that all type parameters are unified as

```
interface Alg_Q<R> extends Alg<R,...,R> {
 Monoid<R> m();

 default R f_j(R p_1, ..., R p_k) {
  return m().join(p_1, m().join(p_2, ..., m().join(
      p_{k-1}, p_k)...)));
 }
 ...
}
```

Figure 8. Generic template for generating boilerplate of queries

type `R`. All arguments to a constructor f_j are combined with `join` from the monoid `m()`. Whereas those arguments with primitive types, like **`int`**, **`boolean`** or `String`, are ignored by default.

5. Generalized Queries

The previous section introduced simple queries where each constructor contributes to a single monoid. Recursive data types, however, often have multiple syntactic categories, for instance expressions and statements. In such multi-sorted Object Algebras each sort is represented by a different type parameter in the algebra interface. In this section we present *generalized queries*, where each such type parameter can be instantiated to different monoids. It turns out that, for some operations, this generalized version of queries is needed.

5.1 Example: data dependencies

A simple example of a generalized query is the extraction of the data dependencies between assignment statements and variables in simple imperative programs. To express this query, the simple `ExpAlg` is first extended with statements using the `StatAlg` interface defined as follows:

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

The extension consists of statement constructors for sequential composition (`Seq`) and assignment (`Assign`). The generated default implementation of queries over statements is shown in Fig. 9, while the generated code for expressions (`G_ExpAlgQuery`) is presented in Appendix A.1.6. Note that the interface declares two monoids, one for each sort. Since the `Assign` and `Seq` constructors create statements, they return elements of the `mStat()` monoid. Furthermore, because it is impossible to automatically join a monoid over one type with a monoid over another type, the `e` argument in `Assign` is ignored. As a result, a concrete implementation has to override this case to deal with the transition from expressions to statements.

Data dependencies are created by assignment statements: for a statement `Assign(String x, Exp e)` method, the variable `x` will depend on all variables appearing in `e`. The result

```
interface G_StatAlgQuery<Exp, Stat>
   extends StatAlg<Exp, Stat> {
 Monoid<Exp> mExp(); Monoid<Stat> mStat();
 default Stat Assign(String x, Exp e) {
  return mStat().empty();
 }
 default Stat Seq(Stat s1, Stat s2) {
  return mStat().join(s1, s2);
 }
}
```

Figure 9. Default implementation of queries over many-sorted statement algebra

```
interface DepGraph extends
   G_ExpAlgQuery<Set<String>>,
   G_StatAlgQuery<Set<String>, Set<Pair<String,
       String>>> {
 default Set<String> Var(String x) {
  return Collections.singleton(x);
 }
 default Set<Pair<String, String>>
          Assign(String x, Set<String> e) {
  Set<Pair<String, String>> deps = new HashSet<>();
  e.forEach(y -> deps.add(new Pair<>(x, y)));
  return deps;
 }
}
```

Figure 10. Dependency Graph with Generalized Query Algebra

of extracting such dependencies can be represented as binary relation (a set of pairs). In expressions we need to collect the free variables, which can be stored in a set of strings. Thus in this traversal two monoids are involved: a monoid for a set of pairs of strings; and a monoid for a set of strings.

To implement the extraction of data dependencies only two cases have to be implemented: the variable (`Var`) case from the `ExpAlg` signature; and the assignment (`Assign`) case from the `StatAlg` signature. The implementation is shown in Fig.10. Note that the `Assign` case takes the input `Set<String> e` and uses it to create the dependency relation. The propagation of dependencies across sequential composition is automatic, as is the propagation of the set of variables through the different types of expressions.

6. Transformations

Queries are a way to extract information from a data structure. Transformations, on the other hand, allow data structures to be changed. Just as with queries, we can distinguish code that deals with traversing the data structure from code that actually changes the structure. In this section we show how to avoid most traversal boilerplate code in the context of transformations based on Object Algebras.

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

Figure 11. A normal algebra-based implementation of variable substitution

6.1 Boilerplate Transformations

A simple example of a transformation algebra, using the object algebra interface `ExpAlg`, is substituting expressions for variables. A manual implementation based on Object Algebras is shown in Fig. 11.

The expression to be substituted, and the variable to substitute for are computed by the methods `e()` and `x()` respectively. `expAlg()` is an instance of `ExpAlg` on which the transformation is based. Since Object Algebras are factories, the transformation is executed immediately during construction. For instance, calling `Var("x")` on a `SubstVar` object with `x()` returning `"x"` immediately returns the result of `e()`, — the original variable expression is never created. In the other cases, the original structure is recreated in the algebra `expAlg()`.

Again we observe the problem of traversal-only boilerplate code: the `Lit` and `Add` methods simply delegate to the base algebra `expAlg()` without performing any computation.

6.2 Generic Traversal Code

The boilerplate code in transformations can be avoided by creating a super-interface containing default methods performing the traversal (shown in Fig. 12). A concrete transformation can then selectively override the cases of interest. Variable substitution can now be implemented as below. In this case, only the method `Var()` is overridden.

```
interface SubstVar<Exp>
   extends ExpAlgTransform<Exp> {
 String x(); Exp e();
 default Exp Var(String s) {
  return s.equals(x())? e(): expAlg().Var(s);
 }
}
```

To execute substitution, the `SubstVar` should be subclassed (either anonymously or explicitly) with implementations for `x()`, `e()`, and `expAlg()`. The base algebra `expAlg()` could for instance be an algebra for pretty printing the expression with the substitution applied. Note that this allows pipelining of transformations: there is no reason `expAlg()` cannot return yet another transformation algebra.

```
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);
 }
}
```

Figure 12. Traversal-only base interface for implementing transformations of expressions

```
interface Alg_T<X_1,...,X_n> extends Alg<X_1,...,X_n> {
 Alg<X_1, ..., X_n> alg();

 default X_i f_j(X_p^1 p_1, ..., X_p^k p_k) {
  return alg().f_j(p_1, ..., p_k);
 }
 ...
}
```

Figure 13. Generic template for generating boilerplate of transformations

Just like in the case of queries, the traversal code in `ExpAlgTransform` is entirely mechanical and can be automatically generated by **Shy**. Annotating algebra interfaces, such as `ExpAlg`, with the annotation `@Algebra`, triggers automatic generation of generic transformation interfaces. Fig. 13 shows the general template for the generated code. Here $\texttt{Alg}_T$ extends `Alg` with the same sorts. And the base algebra `alg()` is declared inside.

7. Contextual Transformations

The previous section introduced a simple template for defining transformations. Transformations in this style may only depend on global context information (e.g., `x()`, `e()`). Many transformations, however, require context information that might change during the traversal itself. In this section we instantiate algebras over function types to obtain transformation which pass information down during traversal. Instead of having the algebra methods delegate directly to base algebra (e.g., `expAlg()`), this now happens indirectly through closures that propagate the context information.

Figure 14 shows the general template for an `Alg<X1, X2, ..., Xn>`, with constructors `f1, ..., fm`. Note that interface $\texttt{Alg}_{CT}$ extends `Alg` and instantiates the type parameters to `Functions` from the context argument `C` to the corresponding sort $\texttt{X}_i$. Each constructor method now creates an anonymous function which, when invoked, calls the functions received as parameters ($\texttt{p}_1$ to $\texttt{p}_k$) and only then creates a structure over the `alg()` algebra.

```
interface Alg_CT<C,X_1,...,X_n>
   extends Alg<Function<C,X_1>,...,Function<C,X_n>> {
  Alg<X_1, ..., X_n> alg();

  default Function<C, X_i> f_j(Function<C,X_p^1> p_1, ...,
      Function<C,X_p^k> p_k) {
    return (c) -> alg().f_j(p_1.apply(c), ..., p_k.apply(
        c));
  }
  ...
}
```

Figure 14. Generic template for generating boilerplate of contextual transformations

```
interface DeBruijn<E> extends
   G_ExpAlgTransform<List<String>, E>,
   G_LamAlgTransform<List<String>, E> {
  default Function<List<String>, E> Var(String p0) {
    return xs ->
       expAlg().Var("" + xs.indexOf(p0) + 1);
  }

  default Function<List<String>, E> Lam(String x,
      Function<List<String>, E> e) {
    return xs ->
       lamAlg().Lam("", e.apply(cons(x, xs)));
  }
}
```

Figure 15. Converting variables to De Bruijn indices

7.1 Example: conversion to De Bruijn indices

An example of a contextual transformation is converting variables to De Bruijn indices in the lambda calculus [6]. Using De Bruijn indices, a variable occurrence is identified by a natural number equal to the number of lambda terms between the variable occurrence and its binding lambda term. Lambda terms expressed using De Bruijn indices are useful because they are invariant with respect to alpha conversion.

To implement the conversion to De Bruijn indices we assume an expression interface `LamAlg` with constructors for lambda abstraction (`Lam`) and application (`App`). This interface can be used together with the `ExpAlg` interface introduced earlier. Furthermore we assume that the generic template shown in Fig. 14 is instantiated for both interfaces as `G_LamAlgTransform` and `G_ExpAlgTransform` respectively. See the full generated code in Appendix A.1.7. Using these interfaces, the conversion to De Bruijn indices can be realized as shown in Fig. 15. Note again that only the relevant cases are overridden: `Var` (from `ExpAlg`) and `Lam` (from `LamAlg`).

8. Extensible Queries and Transformations

Shy queries and transformations inherit modular extensibility from the Object Algebra design pattern. New transformations or queries are simply added by extending the interfaces generated by **Shy**. More interestingly, however, it is also possible to extend the data type with new constructors. Here we briefly describe how queries and transformations can be extended in this case.

Consider again the extension of the expression language with a lambda and application constructs (cf. Section 7). This requires changing the free variables query, since variables bound by `Lam` expressions need to be subtracted from the set of free variables of the body. Instead of reimplementing the query from scratch, it is possible to modularly extend the existing `FreeVars` query:

```
interface FreeVarsWithLambda extends FreeVars,
    LamAlgQuery<Set<String>> {
  default Set<String> Lam(String x,
     Set<String> fv) {
    return fv.stream()
       .filter((y) -> !y.equals(x))
       .collect(toSet());
  }
}
```

The interface `FreeVarsWithLambda` extends both the original `FreeVars` query and the base query implementation that was generated for the `LamAlg` interface defining the language extension. Note again, that only the relevant method (`Lam`) needs to be overridden.

For transformations the pattern is similar. To illustrate extension of transformation, consider the simple transformation that makes all variable occurrences unique. This can be useful to distinguish multiple occurrences of the same name.

```
interface Unique<E> extends ExpAlgTransform<E> {
```

The `Unique` transformation uses a helper method `nextInt()` which returns consecutive integers on each call. The basic transformation simply renames `Var` expressions. If, again, the expression language is extended with lambda constructs, the transformation needs to be updated as well to make the variable in the binding position of lambda expression unique. The following code shows how this can be done in a modular fashion:

```
interface UniqueWithLambda<E> extends Unique<E>,
    LamAlgTransform<E> {
  default E Lam(String x, E e) {
    return lamAlg().Lam(x + nextInt(), e);
  }
}
```

Note that the transformation uses the `lamAlg()` algebra (from `LamAlgTransform`), to create lambda expressions.

Figures 16 and 17 give a high level overview of query and transformation extension using the examples for `FreeVars` and `Unique`, respectively. In the case of queries, the abstract `m()` method will be shared by both the `FreeVars` and `FreeVarsWithLambda` interfaces. On the other hand, transformations are based on multiple base algebras, for sets of data type constructors (e.g., `expAlg()` and `lamAlg()`).

Note finally that, in the current implementation of **Shy** transformations, it is assumed that the language signatures

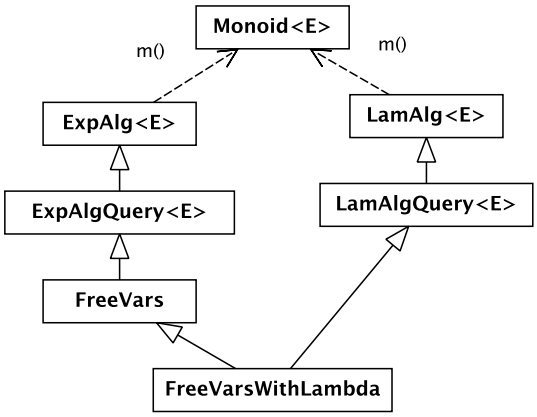

Figure 16. Extension of the `FreeVars` query

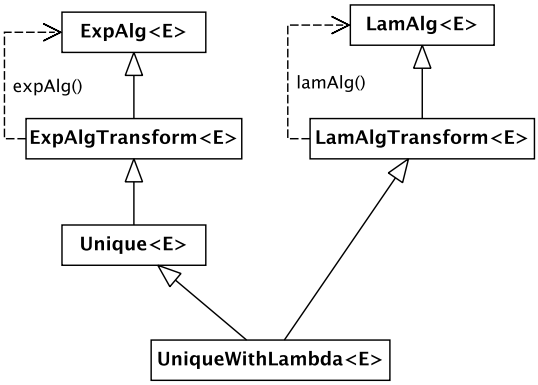

Figure 17. Extension of the `Unique` transformation

`ExpAlg` and `LamAlg` are completely independent. This is however, not an essential requirement. An alternative design could have `LamAlg` be a proper extension of `ExpAlg` (i.e. `LamAlg<E>` **`extends`** `ExpAlg<E>`). In that case, the generated `LamAlgTransform` would need to refine the return type of the `expAlg()` method.

9. Shy Implementation

Shy is implemented using the standard Java annotations framework (`javax.annotation`) packaged in the Java Development Kit. All of the generic traversals discussed in the previous sections – generic queries, generalized generic queries, generic transformations, and contextual generic transformations – are automatically generated by **Shy** for an object algebra interface annotated with `@Algebra`. Furthermore, **Shy** includes the monoid interface as well as several useful instances of the monoid interface.

The big advantage of using standard Java annotations is that the code generation of the generic traversals can be done transparently: users do not need to use or install a tool to generate that code. As a result **Shy** is as simple to use as a conventional library.

Operation	Exp (18)	Stmt (5)	Form (1)
Collect variables	1		
Data dependencies		3	1
Control dependencies		4	1
Type environment		2	
Rename variable	1	2	
Inline conditions		4	
Desugar “unless”		1	
Desugar “repeat”	1	3	

Table 1. Number of overriden cases per query and transformation in the context of the QL implementation

10. Case Study

To illustrate the utility of **Shy** we have implemented a number of queries and transformations in the context of QL, a DSL for questionnaires which has been implemented using Object Algebras before [10]. QL is similar MiniQL, except that it additionally features an if-then-else construct, computed questions (which will appear read only), and a richer expression language. For more information on the features of QL we refer to [7].

10.1 QL Queries and Transformations

The queries extract derived information from a QL program, such as the set of used variables, the data and control dependencies between questions, and the global type environment. The transformations include two transformations of language extensions to the base language. The first realizes a simple desugaring of “unless(c)...” to “if(not(c))...”. The second desugaring statically unfolds a constant bound loop construct (“repeat (i)...”) and renames variables occurring below it accordingly. Finally, we have implemented a simple rename variable operation, and a flattening normalizer which inlines the conditions of nested if-then constructs.

Table 1 shows the number of cases that had to be overridden to implement each particular operation. The second column shows the number of constructs for each sort in QL (Form, Stmt, and Exp). As can be seen, none of the operations required implementing all cases. The bottom row shows the number of overridden cases as a percentage. For this set of queries and transformations, almost no expression cases needed to be overridden, except the “Var” case in collect variables, rename variable and desugar “repeat”[4]. The cases required for desugaring include the case of the language extension (e.g. `Unless` and `Repeat`, respectively). These cases are not counted in the total in the second column but are used to compute the percentage.

[4] Note, however, that the dependency extraction queries reuse the collect variables query on expressions.

10.2 Chaining Transformations

A typical compiler consists of many transformations chained together in a pipeline. **Shy** transformations support this pattern by passing transformation algebras as the base algebra to the implementation of another transformation. For instance, the desugar unless transformation desugars the "unless" statement to "if" statements in another algebra. The latter can represent yet another transformation.

In the context of QL, "unless" desugaring, condition inlining and variable renaming can be chained together as follows:

```
alg = new Desugar<>(new Inline<>(
   new Rename<>(Collections.singletonMap("x", "y"),
     new Format())));
```

The chained transformation `alg` first desugars "unless", then inlines conditions, and finally renames `x`s to `y`s. The `Rename` transformation gets as base algebra an instance of `Format`, a pretty printer for QL.

The algebra `alg` can now be used to create questionnaires:

```
Function<IFormatWithPrecedence, IFormat> pp
  = alg.form("myForm", Arrays.asList(
      alg.unless(alg.var("x"),
      alg.question("x", "X?", new TBoolean()))));
```

The result of constructing this simple questionnaire is a function object representing the "to be inlined" representation of the questionnaire after desugaring. The `IFormatWithPrecedence` and `IFormat` types are formatting operations, respectively representing expressions and statements; these types originate from the `Format` algebra passed to `Rename`. Calling this function with a boolean expression representing **`true`** will trigger inlining of conditions and renaming. The result is then a formatting object (`IFormat`) which can be used to print out the transformed questionnaire:

```
form myForm { if (true && !y) y "X?" boolean }
```

As can be seen, the variable `x` has been renamed to `y`. The (renamed) condition `y` is now negated, because of the desugaring of "unless". Finally, the result of inlining conditions can be observed from the conjunction in the **`if`** statement.

10.3 Shy Performance vs Vanilla ASTs

We compared the performance characteristics of the operations implemented using **Shy** with respect to vanilla implementations based on ordinary AST classes with ordinary methods representing the transformations and queries. The operations were executed on progressively larger QL programs (up to 140Kb). In the vanilla implementation, the program was parsed into an AST structure, and then the operation was invoked and measured. In the case of the **Shy** queries, constructing the "AST" corresponds to executing the query, so we measured that. For context-dependent transformations, however, building the "AST" corresponds to constructing the function to execute the transformation, hence we only measured the execution of invoking this function. The vanilla query implementations use the same monoid structures as in **Shy**.

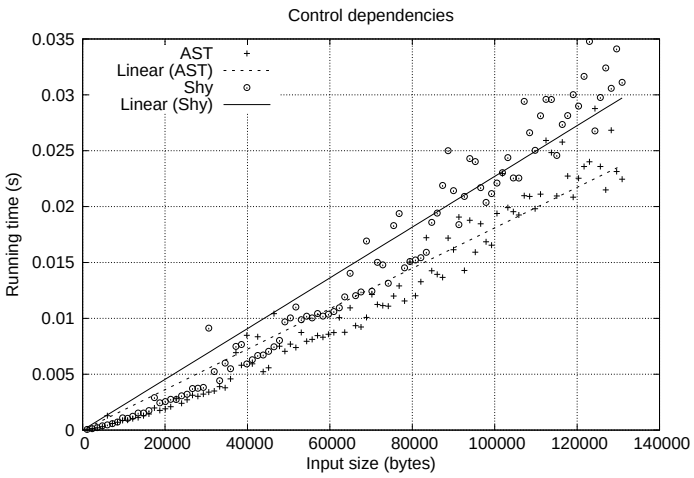

Figure 18. Performance comparison of control dependencies query

The comparison of the control dependencies query is shown in Figure 18. The plot shows that the performance is quite comparable. On average, the **Shy** implementation of the query seems a little slower. This is probably caused by the extensive use of interfaces in the **Shy** framework, whereas the AST-based implementation only uses abstract and concrete classes For transformations the performance difference is slightly more pronounced. Figure 19 shows the performance comparison of the inline conditions transformation. The greater difference can be explained by the fact that creating a new structure in a **Shy** transformation involves dynamically dispatched method calls instead of statically bound constructor calls.

11. Related Work

Structure-shy traversals have been an active research topic. Our approach to structure shy traversal is unique in that it supports separate compilation, modular type checking and data type extension. Furthermore, it can be applied in mainstream languages such as Java. While some approaches support some of these features, to the best of our knowledge, no approach supports all of them. Below we provide a detailed comparison.

HAOYUAN: AOOP [19]

Adaptive Object-Oriented Programming (AOOP) AOOP [19] is an extension of object-oriented programming aimed at increasing the flexibility and maintainability of programs. AOOP promotes the idea of structure-shyness to achieve those goals. In AOOP it is possible to select parts of a structure that should be visited. This is useful to do traversals on

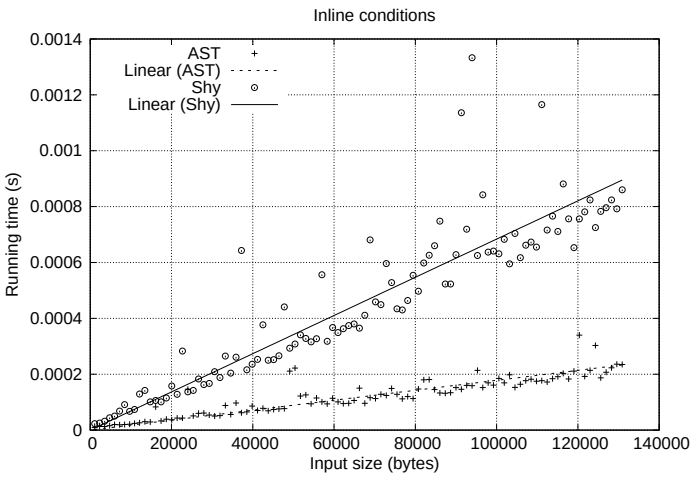

Figure 19. Performance comparison of inline conditions transformation

```
interface AccuTrafoExp<M,E> extends
    ExpAlg<Pair<M,E>> {
  Monoid<M> m(); ExpAlg<E> expAlg();
  default Pair<M,E> Var(String s) {
    return new Pair<>(m().empty(),
       expAlg().Var(s));
  }
  default Pair<M,E> Lit(int i) {
    return new Pair<>(m().empty(),
       expAlg().Lit(i));
  }
  default Pair<M,E> Add(Pair<M,E> e1, Pair<M,E> e2)
  {
    return new Pair<>(m().join(e1.fst(), e2.fst()),
       expAlg().Add(e1.snd(), e2.snd()));
  }
}
```

Figure 20. Combining query and transformation

complex structures and focus only on the interesting parts of the structure relevant for computing the final output. The original approach to AOOP was based on a domain-specific language [19]. DJ is an implementation of AOOP in Java using reflection [24]. More recently DemeterF [5] improved previous approaches to AOOP by providing support for type-safe traversals, generics and data-generic function generation. **Shy** shares with AOOP the use of structure-shyness as a means to increase flexibility and adaptability of programs. Most AOOP approaches, however, are not type-safe. The exception is DemeterF where a custom type system was designed to ensure type-safety of generic functions. Unlike DemeterF, which is a separate language, **Shy** is a Java library. Moreover, compilation of DemeterF programs is implemented through static weaving, and thus appears to preclude separate compilation.

Strategic Programming Strategic programming is an approach to data structure traversal, which originated in term rewriting [2, 28, 29]. Computations are represented by simple, conditional rewrite rules, but the application of such rules is controlled separately using the concept of a strategy. Strategies can be primitive (e.g., "fail") or composed using combinators (e.g., "try s else s'"). The strategy concept has been ported to other paradigms. JJTRAVELER is an OO framework for strategic programming [31]. Lämmel et al. introduced typed strategy combinators in Haskell [16]. The relation between strategic programming and AOOP has been explored in [18].

The key tenet of strategic programming is separation of concerns: actual computation and traversal are specified separately. In **Shy**, the traversal of a data structure is also specified separately (in a super-interface), however, it is fixed for specific styles of queries and transformations. For instance, both queries and transformations employ an innermost, bottom-up strategy.

The distinction between queries and transformations also originates from existing work in strategic programming. Lämmel et al. [16] discuss type unifying and type preserving traversals. Type unifying traversals correspond to our queries, where all data type constructors are unified into a single monoid. Analogously, **Shy** transformations are type preserving in the sense that a transformation is an algebra which maps constructor calls to another algebra of the same type. The ASF+SDF program transformation system distinguishes transforming and accumulating traversals, which correspond to our transformations and queries, respectively. Furthermore, an *accumulating transforming* traversal combines both styles, by tupling the accumulated result and the transformed tree. Support for this combination could easily be generated by **Shy** by having the boilerplate code construct the monoid and the transformed term in parallel (see Fig. 20).

Structure-Shy Traversals with Visitors Visser [31] adapted the strategy combinators of Stratego [29, 30] to combinators that operate on object-oriented Visitors [8]. The resulting framework JJTRAVELER solves the problem of entangling traversal control within the `accept` methods, or in the Visitors themselves (which only allow static specialization). A challenge not addressed by JJTRAVELER is type safety of traversal code: either the combinators needs to be redefined for each data type, or client code needs to cast the generic objects of type `AnyVisitable` to the specific type. Even if specific combinators would be generated, however, the traversed data types would not be extensible.

Another approach to improve upon the standard Visitor pattern is presented in Palsberg et al. [25]. This work particularly addressed the fact that traditional Visitors operate on a fixed set of classes. As a result, the data type

can not be extended without changing all existing Visitors as well. The proposed solution is a generic `Walkabout` class which accesses sub-components of arbitrary data structures using reflection. Unfortunately, the heavy use of reflection make `Walkabouts` significantly slower than traditional Visitors. The authors state that the `Walkabout` class could be generated to improve performance, but note that the addition of a class could trigger regeneration. As a result the pattern does not support separate compilation. Our solution obtains the same kind of default behavior for traversal, without losing extensibility, type safety, or separate compilation.

Whereas the `Walkabout` provides generic navigation over an object structure, this navigation can be programmed explicitly using *guides* [3]. Guides insert one level of indirection between recursing on the children of a node in `visit` methods: the guide decides how to proceed the traversal. Since guides needs to define how to proceed for each type that will be visited, they suffer from the same extensibility problem as ordinary Visitors. Generic guides, on the other hand, are dynamically typed and use reflection to call appropriate `visit` methods. The `Walkabout` can be formulated as such a generic guide. Since the publication of the `Walkabout` [25] Java reflection had improved considerably; nevertheless Bravenboer et al. substantially improved its performance by caching reflective method lookups.

Structure-Shy Traversals in Functional Programming. In functional programming there has been a lot of research on type-safe structure-shy traversals. Lämmel and Peyton Jones' "Scrap your Boilerplate" (SyB) [13–15] series introduced a practical design pattern for doing generic traversals in Haskell. The simple queries and transformations in **Shy** were partly inspired by SyB. However SyB and **Shy** use very different implementation techniques. SyB is implemented in Haskell and relies on a run-time type-safe cast mechanism. This approach allows SyB traversals to be encoded once-and-forall using a single higher-order function called `gfoldl`. In contrast, in **Shy** Java annotations are used to generate generic traversals for each structure. A drawback of SyB traversals is that they are notoriously slow, partly due to the use of the run-time cast [1]. Another notable difference between SyB and **Shy** is with respect to extensibility. While **Shy** supports extensibility of both traversals and structures, the original SyB approach did not support any extensibility. Only in later work, Lämmel and Peyton Jones proposed an alternative design for SyB, based on type classes [33]. This design supports extensibility of traversals, but not of the traversed structures.

In functional programming there has also been an important line of work on *datatype-generic programming* (DGP) [9]. DGP is an advanced form of *generic programming* [20], where generic functions are usually defined by inspecting the structure of types. Different approaches to DGP in Haskell have been extensively studied and documented [11, 27]. DGP can be used to implement structure-shy traversals as combinators, similar to the traversals provided by SyB [12]. Bringert [4] introduced a DGP approach that can also be used to express queries and transformations. Closest to **Shy** is a DGP approach proposed by Lämmel et al. [17] for dealing with so-called "*large bananas*". A large banana corresponds to the fold algebra of a complex structure. Object algebras, which we use in our work, are an OO encoding of fold algebras [21, 22]. However Lämmel et al. work has not dealt with extensibility. Interestingly in their future work Lämmel et al. did mention that they would like "to cope with incomplete or extensible systems of datatypes".

Object Algebras. **Shy** traversals are based on object algebras [21]. The original motivation for object algebras was as a design pattern for OO programming that allowed improved extensibility and modularity of programs. Using object algebras it is possible to solve the well-known "Expression Problem" [32]. Later work [23, 26] has explored the use of *object algebra combinators*, and generalizations of object algebras to improve expressiveness and modularity. In particular it has been claimed that object algebras can be used to do *feature-oriented programming* [23], and to encode *attribute grammars* [26]. One domain where object algebras are especially useful is in the implementation of (extensible) languages. The QL language used in our case study is based on Gouseti et al. [10]. That work provides a realistic implementation of an extensible *domain-specific language* using object algebras. In contrast to our work, previous work on object algebras was mostly motivated by improved programming support for extensibility and modularity. Our work shows that object algebras are also useful to solve a different problem: how to traverse complex structures without boilerplate code. The combination of extensibility (inherited for free from object algebras) and structure-shy type-safe traversals adds a new dimension to our work that, as far as we know, has not been explored previously.

12. Conclusion

This paper showed how various types of traversals for complex structures can be automatically provided by **Shy**. **Shy** traversals are written directly in Java and are type-safe, extensible and separately compilable. There has always been a tension between the correctness guarantees of static typing, and the flexibility of untyped/dynamically-typed approaches. **Shy** shows that even in type systems like Java's, it is possible to get considerable flexibility and adaptability for the problem of boilerplate code in traversals of complex structures, without giving up modular static typing.

There are many avenues for future work. One area of research is to extend **Shy** traversals to support flexible traversal strategies, similarly to strategic programming [2, 28, 29]. Another line of work worth exploring is to adopt generalizations of object algebras [23] for added expressiveness of **Shy** traversals.

References

[1] M. D. Adams and T. M. DuBuisson. Template your boilerplate: Using Template Haskell for efficient generic programming. In *Proceedings of the 2012 ACM SIGPLAN Haskell symposium*, Haskell '12, pages 13–24, 2012.

[2] P. Borovanskỳ, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. *Electronic Notes in Theoretical Computer Science*, 4:35–50, 1996.

[3] M. Bravenboer and E. Visser. Guiding visitors: Separating navigation from computation. Technical Report UU-CS-2001-42, Institute of Information and Computing Sciences, Utrecht University, 2001.

[4] B. Bringert and A. Ranta. A pattern for almost compositional functions. *Journal of Functional Programming*, 18:567–598, September 2008.

[5] B. Chadwick and K. Lieberherr. Weaving generic programming and traversal performance. In *AOSD'10*, 2010.

[6] N. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972.

[7] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, et al. The state of the art in language workbenches. In *Software Language Engineering*. Springer, 2013.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

[9] J. Gibbons. Datatype-generic programming. In R. Backhouse, J. Gibbons, R. Hinze, and J. Jeuring, editors, *Spring School on Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[10] M. Gouseti, C. Peters, and T. v. d. Storm. Extensible language implementation with Object Algebras (short paper). In *GPCE'14*, 2014.

[11] R. Hinze, J. Jeuring, and A. Loeh. Comparing approaches to generic programming in Haskell. In *Datatype-Generic Programming*, volume 4719. Springer Berlin Heidelberg, 2007.

[12] R. Hinze et al. Fun with phantom types. *The Fun of Programming*, pages 245–262, 2003.

[13] R. Lammel and S. P. Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *TLDI'03*, 2003.

[14] R. Lämmel and S. P. Jones. Scrap more boilerplate: Reflection, zips, and generalised casts. In *ICFP '04*, 2004.

[15] R. Lämmel and S. P. Jones. Scrap your boilerplate with class: Extensible generic functions. In *ICFP '05*, 2005.

[16] R. Lämmel and J. Visser. Typed combinators for generic traversal. In *Practical Aspects of Declarative Languages*, pages 137–154. Springer, 2002.

[17] R. Lämmel, J. Visser, and J. Kort. Dealing with large bananas. In J. Jeuring, editor, *Workshop on Generic Programming*, Ponte de Lima, July 2000. Technical Report UU-CS-2000-19, Universiteit Utrecht.

[18] R. Lämmel, E. Visser, and J. Visser. Strategic programming meets adaptive programming. In *AOSD '03*, 2003.

[19] K. J. Lieberherr. *Adaptive Object Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing, 1996.

[20] D. R. Musser and A. A. Stepanov. Generic programming. In *ISAAC '88*, 1989.

[21] B. C. d. S. Oliveira and W. R. Cook. Extensibility for the masses, practical extensibility with object algebras. In *ECOOP '12*, 2004.

[22] B. C. d. S. Oliveira, M. Wang, and J. Gibbons. The visitor pattern as a reusable, generic, type-safe component. In *OOPSLA'08*, 2008.

[23] B. C. d. S. Oliveira, T. van der Storm, A. Loh, and W. R. Cook. Feature-oriented programming with object algebras. In *ECOOP '13*, 2013.

[24] D. Orleans and K. J. Lieberherr. DJ: Dynamic adaptive programming in Java. In *Reflection 2001*. Springer-Verlag, 2001.

[25] J. Palsberg and C. B. Jay. The essence of the visitor pattern. In *COMPSAC'98*, 1998.

[26] T. Rendel, J. I. Brachthäuser, and K. Ostermann. From object algebras to attribute grammars. In *OOPSLA '14*, 2014.

[27] A. Rodriguez, J. Jeuring, P. Jansson, A. Gerdes, O. Kiselyov, and B. C. d. S. Oliveira. Comparing libraries for generic programming in Haskell. In *Haskell'08*, 2008.

[28] M. G. J. van den Brand, P. Klint, and J. J. Vinju. Term rewriting with traversal functions. *ACM Trans. Softw. Eng. Methodol.*, 12(2):152–190, Apr. 2003.

[29] E. Visser and Z.-e.-A. Benaissa. A core language for rewriting. *Electronic Notes in Theoretical Computer Science*, 15: 422–441, 1998.

[30] E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *ICFP '98*, 1998.

[31] J. Visser. Visitor combination and traversal control. In *OOPSLA '01*, 2001.

[32] P. Wadler. The Expression Problem. Email, Nov. 1998. Discussion on the Java Genericity mailing list.

[33] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89*, 1989.

A. Appendix

A.1 Complete Code

A.1.1 OO Approach for `usedVars` and `rename`

Below is the complete code for Fig. **??**, which implements `usedVars` and `rename` in the QL example, as an OO approach.

```
class Form {
 String name;
 List<Stmt> body;
 Form(String id, List<Stmt> body) {
  this.name = id;
  this.body = new ArrayList<Stmt>(body);
 }
 Set<String> usedVars() {
  Set<String> vars = new HashSet<>();
  body.forEach(s -> vars.addAll(s.usedVars()));
  return vars;
 }
 Form rename() {
  List<Stmt> ss = new ArrayList<>();
  for (Stmt s: body) ss.add(s.rename());
  return new Form(name, ss);
 }
}

abstract class Stmt {
 abstract Set<String> usedVars();
 abstract Stmt rename();
}

class If extends Stmt {
 Exp cond;
 Stmt then;
 If(Exp cond, Stmt then) {
  this.cond = cond;
  this.then = then;
 }
 Set<String> usedVars() {
  Set<String> vars = new HashSet<>(cond.usedVars())
      ;
  vars.addAll(then.usedVars());
  return vars;
 }
 If rename() {
  return new If(cond.rename(), then.rename());
 }
}

class Question extends Stmt {
 String name, label, type;
 Question(String n, String l, String t) {
  this.name = n;
  this.label = l;
  this.type = t;
 }
 Set<String> usedVars() {
  return emptySet();
 }
 Question rename() {
  return new Question(name + "_", label, type);
 }
}


abstract class Exp {
 abstract Set<String> usedVars();
 abstract Exp rename();
}

class Lit extends Exp {
 int n;
 Lit(int n) {
  this.n = n;
 }
 Set<String> usedVars() {
  return emptySet();
 }
 Lit rename() {
  return new Lit(n);
 }
}

class Var extends Exp {
 String x;
 Var(String name) {
  this.x = name;
 }
 Set<String> usedVars() {
  return Collections.singleton(x);
 }
 Var rename() {
  return new Var(x + "_");
 }
}

class GEq extends Exp {
 Exp lhs, rhs;
 GEq(Exp lhs, Exp rhs) {
  this.lhs = lhs;
  this.rhs = rhs;
 }
 Set<String> usedVars() {
  Set<String> vars = new HashSet<>(lhs.usedVars());
  vars.addAll(rhs.usedVars());
  return vars;
 }
 GEq rename() {
  return new GEq(lhs.rename(), rhs.rename());
 }
}
```

A.1.2 `UsedVars` implementing the `QLAlg` interface

Below is the class `UsedVars` implementing `QLAlg` by hand, which is the QL example encoded by object algebras. See Section 3.2.

```
class UsedVars implements
  QLAlg<Set<String>, Set<String>, Set<String>> {

 public Set<String> Form(String n,
               List<Set<String>> b) {
  Set<String> vars = new HashSet<>();
  b.forEach(s -> vars.addAll(s));
  return vars;
 }

 public Set<String> If(Set<String> c, Set<String> t
     ) {
  Set<String> vars = new HashSet<>(c);
  vars.addAll(t);
```

```
    return vars;
  }

  public Set<String> Question(String n, String l,
      String t) {
    return Collections.emptySet();
  }

  public Set<String> Lit(int x) {
    return Collections.emptySet();
  }

  public Set<String> Var(String x) {
    return Collections.singleton(x);
  }

  public Set<String> GEq(Set<String> l, Set<String>
      r) {
    Set<String> vars = new HashSet<>(l);
    vars.addAll(r);
    return vars;
  }
}
```

A.1.3 `Rename` implementing the `QLAlg` interface

Similarly, the following code gives the implementation of `Rename` that implements `QLAlg` in Section 3.2.

```
class Rename<E, S, F> implements QLAlg<E, S, F> {
  private QLAlg<E, S, F> alg;
  public Rename(QLAlg<E, S, F> alg) {
    this.alg = alg;
  }
  public F Form(String id, List<S> stmts) {
    return alg.Form(id, stmts);
  }
  public S If(E cond, S then) {
    return alg.If(cond, then);
  }
  public S Question(String id, String lbl,
     String type) {
    return alg.Question(id + "_", lbl, type);
  }
  public E Lit(int x) { return alg.Lit(x); }
  public E Var(String name) {
    return alg.Var(name + "_");
  }
  public E GEq(E lhs, E rhs) {
    return alg.GEq(lhs, rhs);
  }
}
```

A.1.4 `QLAlgQuery`: generated code

The generated code for `QLAlgQuery` by **Shy** in Fig. **??**.

```
public interface QLAlgQuery<R>
   extends QLAlg<R, R, R> {

  Monoid<R> m();

  default R Form(java.lang.String p0,
     java.util.List<R> p1) {
    R res = m().empty();
    res = m().join(res, m().fold(p1));
    return res;
  }

  default R Geq(R p0, R p1) {
    R res = m().empty();
    res = m().join(res, p0);
    res = m().join(res, p1);
    return res;
  }

  default R If(R p0, R p1) {
    R res = m().empty();
    res = m().join(res, p0);
    res = m().join(res, p1);
    return res;
  }

  default R Lit(int p0) {
    R res = m().empty();
    return res;
  }

  default R Question(java.lang.String p0,
     java.lang.String p1, java.lang.String p2) {
    R res = m().empty();
    return res;
  }

  default R Var(java.lang.String p0) {
    R res = m().empty();
    return res;
  }

}
```

A.1.5 `QLAlgTransform` and `QLAlgTrans`: generated code

The code for `QLAlgTransform` and its class representation `QLAlgTrans` for use, generated by **Shy**. See Fig. **??**.

```
public interface QLAlgTransform<A0, A1, A2>
   extends QLAlg<A0, A1, A2> {

  QLAlg<A0, A1, A2> qLAlg();

  @Override
  default A2 Form(java.lang.String p0,
     java.util.List<A1> p1) {
    return qLAlg().Form(p0, p1);
  }

  @Override
  default A0 Geq(A0 p0, A0 p1) {
    return qLAlg().Geq(p0, p1);
  }

  @Override
  default A1 If(A0 p0, A1 p1) {
    return qLAlg().If(p0, p1);
  }

  @Override
  default A0 Lit(int p0) {
    return qLAlg().Lit(p0);
  }
```

```
  @Override
  default A1 Question(java.lang.String p0,
     java.lang.String p1, java.lang.String p2) {
    return qLAlg().Question(p0, p1, p2);
  }

  @Override
  default A0 Var(java.lang.String p0) {
    return qLAlg().Var(p0);
  }

}

public class QLAlgTrans<A0, A1, A2>
   implements QLAlgTransform<A0, A1, A2> {

  private QLAlg<A0, A1, A2> alg;

  public QLAlgTrans(QLAlg<A0, A1, A2> alg) {
    this.alg = alg;
  }

  public QLAlg<A0, A1, A2> qLAlg() {return alg;}

}
```

A.1.6 `G_ExpAlgQuery`: generated code

The generated code for `G_ExpAlgQuery` by **Shy** in Fig. 10.

```
public interface G_ExpAlgQuery<A0>
   extends ExpAlg<A0> {

  Monoid<A0> mExp();

  @Override
  default A0 Add(A0 p0, A0 p1) {
    A0 res = mExp().empty();
    res = mExp().join(res, p0);
    res = mExp().join(res, p1);
    return res;
  }

  @Override
  default A0 Lit(int p0) {
    A0 res = mExp().empty();
    return res;
  }

  @Override
  default A0 Var(java.lang.String p0) {
    A0 res = mExp().empty();
    return res;
  }

}
```

A.1.7 `G_ExpAlgTransform` and `G_LamAlgTransform`: generated code

The generated code for `G_ExpAlgTransform` and `G_LamAlgTransform` by **Shy** in Fig. 15.

```
public interface G_ExpAlgTransform<A, B0>
   extends ExpAlg<Function<A, B0>> {

  ExpAlg<B0> expAlg();

  default <B> List<B> substListExpAlg(List<Function<
      A, B>> list, A acc) {
    List<B> res = new ArrayList<B>();
    for (Function<A, B> i : list)
      res.add(i.apply(acc));
    return res;
  }

  @Override
  default Function<A, B0> Add(Function<A, B0> p0,
      Function<A, B0> p1) {
    return acc -> expAlg().Add(p0.apply(acc),
       p1.apply(acc));
  }

  @Override
  default Function<A, B0> Lit(int p0) {
    return acc -> expAlg().Lit(p0);
  }

  @Override
  default Function<A, B0> Var(java.lang.String p0) {
    return acc -> expAlg().Var(p0);
  }

}

public interface G_LamAlgTransform<A, B0>
   extends LamAlg<Function<A, B0>> {

  LamAlg<B0> lamAlg();

  default <B> List<B> substListLamAlg(List<Function<
      A, B>> list, A acc) {
    List<B> res = new ArrayList<B>();
    for (Function<A, B> i : list)
      res.add(i.apply(acc));
    return res;
  }

  @Override
  default Function<A, B0> Apply(Function<A, B0> p0,
      Function<A, B0> p1) {
    return acc -> lamAlg().Apply(p0.apply(acc),
       p1.apply(acc));
  }

  @Override
  default Function<A, B0> Lam(java.lang.String p0,
      Function<A, B0> p1) {
    return acc -> lamAlg().Lam(p0, p1.apply(acc));
  }

}
```