Scrap your Boilerplate with Object Algebras

Name1
Affiliation1
Email1

Name2 Name3
Affiliation2/3
Email2/3

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 *extensible* data structures. Using Java annotations generic boilerplate code is generated for various types of traversals, including queries and transformations. Thus, using **Shy**, 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. Moreover, traversals using the **Shy** framework are also much more *structure shy*, becoming more adaptive to future changes or extensions to 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.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

Many applications require complex recursive data structures. Examples abound, for example, in language processing tools/libraries for programming languages, domain-specific languages, mark-up 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 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 structures. Static types formalize the distinction 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, when traversing such structures, the number of nodes and the enforced type distinctions between nodes can lead to so-called *boilerplate code* [14]: code that is similar for most types of nodes and which essentially just walks the structure. Traversals where such boilerplate code dominates are called *structure shy* [20]. In structure shy operations the "interesting" code is limited to only a small portion of nodes. 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 data structures 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) [20] 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. 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 a change to the traversed data type only affects the generated boilerplate code, the user-defined code can remain unchanged. 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" [14] 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* [22] 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 algebra interfaces are combined with Java annotations to generate generic and reusable object algebras that deal with boilerplate traversal code. Those object algebras include different types of traversals: *queries*; *transformations*; *generalized queries*; and *contextual transformations*. Programmers who want to implement structure-shy traversals can inherit the generic traversal code, and focus only on writing the interesting parts of the traversal. Consequently, the amount of code that programmers need to write is significantly smaller. 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 or extensions to the data type.
- **Simple and general: Shy** traversals work for any structure that can be expressed as a (multi-sorted) object algebra. This includes complex OO hierarchies or ASTs for large languages. Very often traversals are quite simple, being implementable in just a few lines of code, even for complex structures with hundreds of different types of nodes.
- **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-safe with separate compilation: 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. Furthermore **Shy** traversals support separate compilation.
- **Extensible:** Traversals inherit type-safe extensibility from object algebras. Both traversals and structures are extensible. Therefore it is possible to reuse traversal code in structures that are extended with additional node types.

To prove the effectiveness of the approach, we have applied **Shy** in the implementation of the domain-specific questionnaire language QL [11]. Our results show that for a large number of traversals there was a significant reduction in the amount of user-defined code.

Although we have chosen Java as the implementation language for **Shy**, our approach should apply to any OO language with support for generics and annotations. For example it should be easy to port **Shy** to languages such as 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 have implemented[1] 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.
- **Case study and empirical evaluation.** We evaluate the approach 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.

2. Overview

This section starts by motivating the problem of boilerplate code in traversals of complex structures. It then shows how **Shy** addresses the problem using a combination of object algebras [22] and Java annotations.

2.1 A Company Structure and an Object Oriented Solution

We start by considering the company structure introduced in Fig. 1. The example is inspired by Lämmel and Peyton Jones company structure, used to motivate their "Scrap your Boilerplate" approach [14]. It is also closely related to the 101 companies system [8]. A `Company` comprises a list of `Departments`. Each `Department` is managed by an `Employee`, and contains a list of `Units`. A `Unit` can be either a `Department` or an `Employee`. An `Employee` has a `Person` and `Salary` information.

[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.

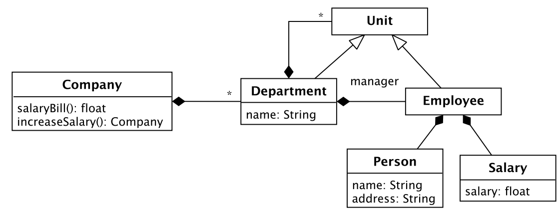


Figure 1. Variant of the 101 Companies data model

```
class Company {
  private List<Department> depts;
  Company(List<Department> depts) { this.depts = depts; }
  Float salaryBill() {
    return depts.stream().reduce(0.0f, (s, x) -> s + x.
         salaryBill(),
                                (x, y) -> x + y);
  }
  Company increaseSalary() {
    return new Company(depts.stream().map(d -> d.
         increaseSalary())
                              .collect(toList()));
  }
}
class Salary {
  private Float salary;
  public Salary(Float salary) { this.salary = salary; }
  public Float salaryBill() { return this.salary; }
  public Salary increaseSalary() { return new Salary(this.
       salary * 1.1f); }
}
```

Figure 2. The `Company` and `Salary` classes implementing part of the company structure.

Two operations are supported by the company structure: querying the salary bill for the whole company; and increasing the salary of each employee by 10%. To support those operations, an OO solution is to have methods `salaryBill`, which returns salary as a float number, and `increaseSalary`, which constructs a company with updated salaries.

Fig. 2 shows the Java implementation of the `Salary` and `Company` classes. The implementation of other classes is omitted for space reasons. In the implementation, querying the salary bill of the whole company is done by returning the salary in instances of the `Salary` class, and delegating the method `salaryBill` to the child nodes in other classes. Increasing the salary for the whole company by 10%, following Lämmel and Peyton Jones approach, can be implemented similarly. The `Salary` class will construct a new `Salary` object with increased salary information, and the method `increaseSalary` in other classes will construct corresponding new objects with updated salaries.

However, this simple OO solution has two important problems:

1. **Lack of extensibility** The OO style representation of data structures is cumbersome and inflexible due to the bound relationship between classes. For instance, adding a new operation such as pretty printing the company structure requires pervasive changes across all existing classes.
2. **Boilerplate traversal code** Another issue is that we usually need to write a large amount of boilerplate code to implement traversals. In our example, we implemented `increaseSalary` and `salaryBill` in all classes. However only the `Salary` class

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

Figure 3. Company structure represented by an object algebra interface

does some interesting work. All other classes simply delegate the task to the child nodes. This problem becomes more severe with bigger tree structures. The interesting code can become a very small portion of the code needed to perform a traversal: most of the code is doing tedious walking of the structure.

2.2 Modeling the Company Structure with Object Algebras

Object algebras [22] provide a solution to the extensibility problem. Object algebras are a design pattern that is useful to model complex structures, while providing a lot of flexibility in terms of modularity and extensibility. In particular, object algebras allow extensibility in two dimensions: it is easy to add new types of nodes; and new operations over the structure.

Fig. 3 shows the approach to model the company structure as an object algebra interface. Each kind of node (for example `Salary` or `Company`) is represented as a type parameter. Each method in the object algebra interface models a constructor of some node in the company structure. For example the method `C` constructs an instance of a company; whereas the method `S` creates an instance of a salary. From an object-oriented perspective the methods in the interface are factory methods. For the reader familiar with functional programming, the resemblance to constructors of (a system of) algebraic data types should be clear.

Operations are defined by implementing the object algebra interface. The following code show a partial implementation of the salary bill operation.

```
class SalaryBill implements OneOhOneAlg<Float, ..., Float>
      {
  public Float C(List<Float> depts) {
    return depts.stream().reduce(0.0f, (x, y) -> x + y);
  }
  ...
  public Float S(float salary) { return salary; }
}
```

The `SalaryBill` class provides an implementation for each of the methods in the object algebra interface, which defines the overall salary bill operation. Since the result of computing the salary bill is a `Float`, all the type parameters are set to that type. Most of the method implementations simply traverse the child nodes and accumulate the salaries. That is the case, for example, for `C`. The only method implementation that does something different is `S`, which returns the salary argument.

For the increase salary operation the result is itself a structure with all salaries updated. The following code shows part of the implementation:

```
class IncreaseSalary<C,D,U,E,P,S> implements OneOhOneAlg<C
      ,D,U,E,P,S> {
  private OneOhOneAlg<C,D,U,E,P,S> alg;
  public Float C(List<Dept> depts) { return alg.C(depts); }
  ...
  public Salary S(float salary) { return alg.S(1.1f *
        salary); }
}
```

```
class SalaryBill implements OneOhOneAlgQuery<Float> {
  public Monoid<Float> m() {return new FloatMonoid();}
  public Float S(float sal) {return sal;}
}

class IncreaseSalary<C,D,U,E,P,S> extends OneOhOneAlgTrans
      <C,D,U,E,P,S> {
  IncreaseSalary(OneOhOneAlg<C,D,U,E,P,S> alg) { super(alg)
        ; }
  public S S(float salary) { return oneOhOneAlg().S(1.1f *
        salary); }
}
```

Figure 4. Salary bill and increase salary with **Shy**.

The class `IncreaseSalary` is parameterized by six types, which represent the kinds of nodes in the company structure. Each constructor needs to build an instance of the right type of nodes. Note that, due to the lack of better type-inference in Java, there is some repetition of type-annotations. In order to create the updated company structure another algebra called `alg` is used. Almost all the method implementations reconstruct the structure with no changes using the methods in `alg`. The exception is the method `S` where the structure is also reconstructed, but the salary is increased by 10%.

Although we solved the problem of extensibility with object algebras, the traversal code is still lengthy and we are still writing tedious traversal code. In both operations, the only interesting code is in the `S` method. Ideally, we would like to write only the code for the interesting cases, and somehow "inherit" the remainder tedious traversal code.

2.3 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 generic traversals at the cost of a simple 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*. Those two types of traversals are useful to capture, respectively, the salary bill and increase salary operations.

With **Shy**, programmers just need to add the `@Algebra` annotation to the definition of `OneOhOneAlg` to get the code for generic queries and transformations. An example of that annotation is already shown in Fig. 3. With that annotation several classes are generated automatically, including `OneOhOneAlgQuery` and `OneOhOneAlgTrans`. Using those classes the code needed to write the salary bill and increase salary object algebras is much shorter. The code for the two operations is shown in Fig. 4. By implementing the `OneOhOneAlgQuery` and `OneOhOneAlgTrans` classes, only the `S` method needs to be overridden: all the other methods, which do a simple generic traversal, are inherited. 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 the transformation code.

Some client code is shown as follows:

```
<C,D,U,E,P,S> C makeCompany(OneOhOneAlg<C,D,U,E,P,S> alg)
      {
  E ralf = alg.E(alg.P("Ralf", "Amsterdam"), alg.S(8000.0f)
        );
  E joost = alg.E(alg.P("Joost", "Amsterdam"), alg.S(1000.0
        f));
  E simon = alg.E(alg.P("Simon", "Cambridge"), alg.S(2000.0
        f));
  E blair = alg.E(alg.P("Blair", "London"), alg.S(100000.0f
        ));
  List<U> us = asList(alg.PU(joost), alg.PU(simon));
```

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

Figure 5. The ExpAlg object algebra interface.

```
  List<D> ds = asList(alg.D("Research", ralf, us),
                     alg.D("Strategy", blair, emptyList()));
  return alg.C(ds);
}
```

This method is used to create a particular company structure, in the usual object algebras style. Using this company structure, we can use the `SalaryBill` and `IncreaseSalary` to compute some information about the company:

```
SalaryBill salaryBill = new SalaryBill();
System.out.println(makeCompany(salaryBill));
IncreaseSalary<Float,Float,Float,Float,Float,Float>
      incSalary
  = new IncreaseSalary<>(salaryBill);
System.out.println(makeCompany(incSalary));
```

The results give `111000.0` and `122100.0`, which is respectively the salary before and after increase.

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*.

3. 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" [14].

Fig. 5 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.

3.1 Boilerplate Queries

Fig. 6 shows a standard approach for computing free variables using object algebras[2]. 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.

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

[2] 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 singleton(s);
       }
  default Set<String> Lit(int i) { return emptySet(); }
  default Set<String> Add(Set<String> e1, Set<String> e2) {
    return concat(e1.stream(), e2.stream()).collect(toSet()
         );
  }
}
```

Figure 6. Free variables as an object algebra.

```
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.

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.

3.2 Abstracting Generic Traversal Code

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".

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.

3.3 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>(); }
```

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

  default R fj(R p1, ..., R pk) {
    return m().join(p1, m().join(p2, ..., m().join(pk−1, pk)
         ...)));
  }
  ...
}
```

Figure 8. Generic template for generating boilerplate of queries

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

Figure 9. Statement Algebra Interface

```
  default Set<String> Var(String s) { return 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 type `R`. All arguments to a constructor f_j are combined with `join` from the monoid `m()`.

4. 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.

4.1 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 of Fig. 9. 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. 10. 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 of extracting such

```
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 10. 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 singleton(x);
      }
  default Set<Pair<String, String>> Assign(String x, Set<
      String> e) {
    return e.stream().map(y -> new Pair<>(x, y)).collect(
        toSet());
  }
}
```

Figure 11. Dependency Graph with Generalized Query Algebra

```
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 12. A normal algebra-based implementation of variable substitution

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.11. 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.

5. 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.

5.1 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. 12.

```
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 13. Traversal-only base interface for implementing transformations of expressions

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

Figure 14. Implementation of variable substitution using **Shy**

```
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 15. Generic template for generating boilerplate of transformations

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 `exp()`, — 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.

5.2 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. 13). A concrete transformation can then selectively override the cases of interest. Variable substitution can now be implemented as shown in Fig. 14. In this case, only the method `Var()` is overridden.

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.

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. 15 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.

```
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 16. 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 17. Converting variables to De Bruijn indices

6. 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 16 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.

6.1 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. 16 is instantiated for both interfaces as `G_LamAlgTransform` and `G_ExpAlgTransform` respectively. Using these interfaces, the conversion to De Bruijn indices can be realized as shown in Fig. 17. Note again that only the relevant cases are overridden: `Var` (from `ExpAlg`) and `Lam` (from `LamAlg`).

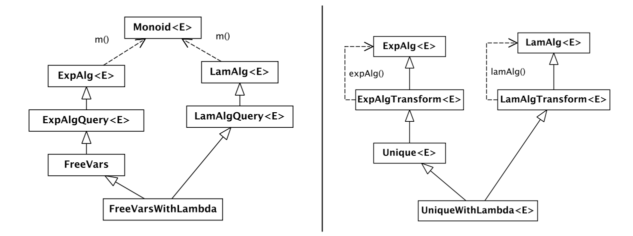


Figure 18. Extension of the `FreeVars` query (left) and the `Unique` transformation (right)

7. Extensible Queries and Transformations

Shy queries and transformation 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 6). 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> {
  int nextInt();
  default E Var(String s) { return expAlg().Var(s + nextInt
      ()); }
}
```

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.

Figure 18 gives a high level overview of query and transformation extension using the examples for `FreeVars` and `Unique`. 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 `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.

8. 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.

9. Case Study

To illustrate the utility of **Shy** we have implemented a number of queries and transformations in the context of QL, a simple DSL for questionnaires implemented using Object Algebras [11]. A questionnaire is rendered as an interactive form where, depending on user actions, new questions may appear, or values may be computed. An example QL questionnaire is shown on the left of Fig. 19 together with its rendering on the right.

QL programs consist of lists of labeled, typed questions. Questions can be answerable, meaning the user has to enter some data, or computed, in which case the question is defined by an expression. A conditional if-then-else construct allows questions to visually appear only when a certain condition is true.

9.1 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 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"[3]. 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.

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

```
form HouseOwning {
  soldHouse: "Did you sell a
      house?" boolean
  boughtHouse: "Did you buy a
      house?" boolean
  if (soldHouse) {
    sellingPrice: "Selling
        price:" integer
    privateDebt: "Private debts
        :" integer
    valueResidue: "Value
        residue:" integer
      = (sellingPrice -
          privateDebt)
  }
}
```

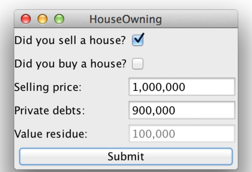


Figure 19. Example QL questionnaire (left) and its rendering (right)

Sort	#Cases	Collect vars	Data deps	Control deps	Type env	Rename var	Inline conds	unless	repeat
Form	1		1	1					
Stmt	5		3	4	2	2	4	1	3
Exp	18	1				1			1
		4%	17%	21%	8%	13%	17%	4%	16%

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

9.2 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<>(ren, new
      Format())));
```

The chained transformation `alg` first desugars "unless", then inlines conditions, and finally renames variables according to the map `ren`. 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", 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.

9.3 Shy Performance vs Vanilla ASTs

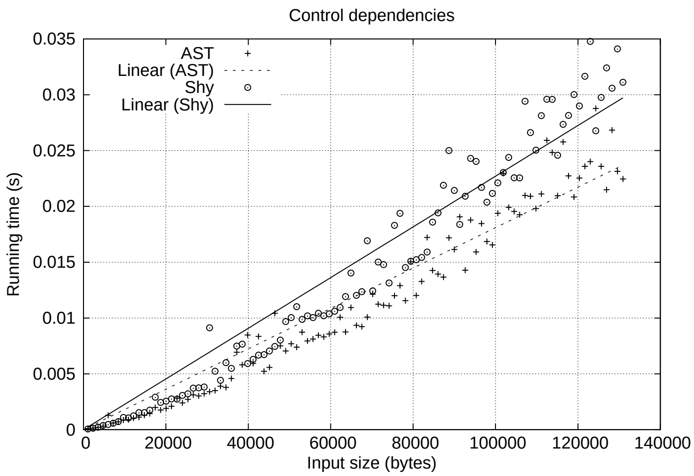
We compared the performance characteristics of the operations implemented using **Shy** with respect to vanilla implementations based on ordinary AST classes with ordinary methods representing the transformations and queries. The operations were executed on progressively larger QL programs (up to 140Kb). 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. To make the comparison as fair as possible, the vanilla query implementations use the same monoid structures as in **Shy**.

The comparison of the control dependencies query is shown on the left of Fig. 20. 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. The right of Fig. 20 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.

10. 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.

Adaptive Object-Oriented Programming (AOOP) AOOP [20] 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 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 [20]. DJ is an implementation of AOOP in Java using reflection [25]. More recently DemeterF [7] improved previ-



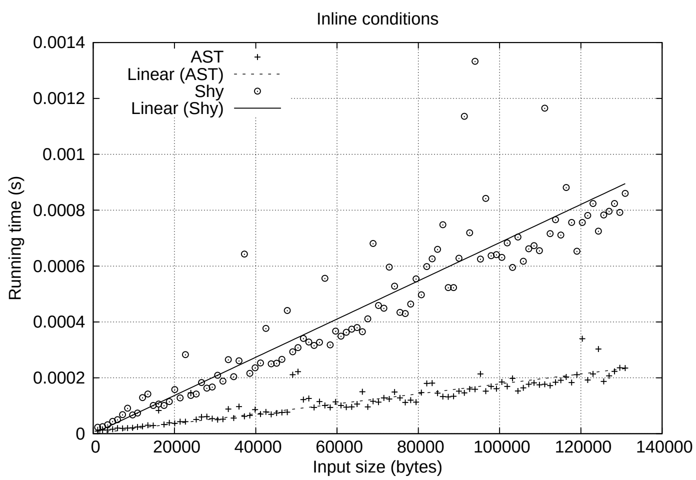


Figure 20. Performance comparison of control dependencies query (left) and inline conditions transformation (right) implemented using normal ASTs vs using the **Shy** framework

ous 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 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, 3, 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 strat-

```
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 21. Combining query and transformation

egy 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 [18]. The relation between strategic programming and AOOP has been explored in [17].

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. [18] 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. 21).

Structure-Shy Traversals with Visitors Visser [31] adapted the strategy combinators of Stratego [29, 30] to combinators that operate on object-oriented Visitors [9]. 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. [26]. 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* [4]. 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` [26] 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) [14–16] 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 [32]. 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) [10]. DGP is an advanced form of *generic programming* [21], where generic functions are usually defined by inspecting the structure of types. Different approaches to DGP in Haskell have been extensively studied and documented [12, 28]. DGP can be used to implement structure-shy traversals as combinators, similar to the traversals provided by SyB [13]. Bringert [5] 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. [19] 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 [22, 24]. 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 [22]. 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" [33]. Later work [23, 27] 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* [27]. 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. [11]. 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.

11. 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 of avenues for future work. One area of research is to extend **Shy** traversals to support flexible traversal strategies, similarly to strategic programming [2, 3, 29]. Another line of work worth exploring is to adopt generalizations of object algebras [23] for added expressiveness of **Shy** traversals.

References

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

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

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

[4] Bravenboer, M., Visser, E.: Guiding visitors: Separating navigation from computation. Tech. Rep. UU-CS-2001-42, Institute of Information and Computing Sciences, Utrecht University (2001)

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

[6] de Bruijn, N.: 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] Chadwick, B., Lieberherr, K.: Weaving generic programming and traversal performance. In: AOSD'10 (2010)

[8] Favre, J.M., Lämmel, R., Schmorleiz, T., Varanovich, A.: *101companies*: a community project on software technologies and software languages. In: Proceedings of TOOLS 2012. LNCS, Springer (2012)

[9] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)

[10] Gibbons, J.: Datatype-generic programming. In: Backhouse, R., Gibbons, J., Hinze, R., Jeuring, J. (eds.) Spring School on Datatype-Generic Programming. Lecture Notes in Computer Science, vol. 4719. Springer-Verlag (2007)

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

[12] Hinze, R., Jeuring, J., Loeh, A.: Comparing approaches to generic programming in Haskell. In: Datatype-Generic Programming, vol. 4719. Springer Berlin Heidelberg (2007)

[13] Hinze, R., et al.: Fun with phantom types. The Fun of Programming pp. 245–262 (2003)

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

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

[16] Lämmel, R., Jones, S.P.: Scrap your boilerplate with class: Extensible generic functions (2005)

[17] Lämmel, R., Visser, E., Visser, J.: Strategic programming meets adaptive programming. In: AOSD '03 (2003)

[18] Lämmel, R., Visser, J.: Typed combinators for generic traversal. In: Practical Aspects of Declarative Languages, pp. 137–154. Springer (2002)

[19] Lämmel, R., Visser, J., Kort, J.: Dealing with large bananas. In: Jeuring, J. (ed.) Workshop on Generic Programming. Technical Report UU-CS-2000-19, Universiteit Utrecht, Ponte de Lima (Jul 2000)

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

[21] Musser, D.R., Stepanov, A.A.: Generic programming. In: ISAAC '88 (1989)

[22] Oliveira, B.C.d.S., Cook, W.R.: Extensibility for the masses, practical extensibility with object algebras. In: ECOOP '12 (2004)

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

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

[25] Orleans, D., Lieberherr, K.J.: DJ: Dynamic adaptive programming in Java. In: Reflection 2001. Springer-Verlag (2001)

[26] Palsberg, J., Jay, C.B.: The essence of the visitor pattern. In: COMPSAC'98 (1998)

[27] Rendel, T., Brachthäuser, J.I., Ostermann, K.: From object algebras to attribute grammars. In: OOPSLA '14 (2014)

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

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

[30] Visser, E., Benaissa, Z.e.A., Tolmach, A.: Building program optimizers with rewriting strategies. In: ICFP '98 (1998)

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

[32] Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: POPL '89 (1989)

[33] Wadler, P.: The Expression Problem. Email (Nov 1998), discussion on the Java Genericity mailing list