Extensibility for the Masses
Practical Extensibility with Object Algebras

Bruno C. d. S. Oliveira[1] and William R. Cook[2]

[1] Seoul National University
bruno@ropas.snu.ac.kr
[2] University of Texas, Austin
wcook@cs.utexas.edu

Abstract. This paper presents a new solution to the expression problem (EP) that works in OO languages with simple generics (including Java or C#). A key novelty of this solution is that advanced typing features, including F-bounded quantification, wildcards and variance annotations, are not needed. The solution is based on *object algebras*, which are an abstraction closely related to algebraic datatypes and Church encodings. Object algebras also have much in common with the traditional forms of the Visitor pattern, but without many of its drawbacks: they are extensible, remove the need for accept methods, and do not compromise encapsulation. We show applications of object algebras that go beyond toy examples usually presented in solutions for the expression problem. In the paper we develop an increasingly more complex set of features for a mini-imperative language, and we discuss a real-world application of object algebras in an implementation of remote batches. We believe that object algebras bring extensibility to the masses: object algebras work in mainstream OO languages, and they significantly reduce the conceptual overhead by using only features that are used by everyday programmers.

1 Introduction

The "expression problem" (EP) [37, 10, 45] is now a classical problem in programming languages. It refers to the difficulty of writing data abstractions that can be easily extended with both new operations and new data variants. Traditionally the kinds of data abstraction found in functional languages can be extended with new operations, but adding new data variants is difficult. The traditional object-oriented approach to data abstraction facilitates adding new data variants (classes), while adding new operations is more difficult. The Visitor Pattern [13] is often used to allow operations to be added to object-oriented data abstractions, but the common approach to visitors prevents adding new classes. Extensible visitors can be created [42, 49, 30], but so far solutions in the literature require complex and unwieldy types, or advanced programming languages.

In this paper we present a new approach to the EP based on *object algebras*. An object algebra is a class that implements a generic abstract factory interface, which corresponds to a particular kind of *algebraic signature* [17]. Object

algebras are closely related to the Abstract Factory, Builder and Visitor patterns and can offer improvements on those patterns. Object algebras have strong theoretical foundations, inspired by earlier work on the relation between Church encodings and the Visitor pattern [5, 29, 34].

Object algebras use simple, intuitive generic types that work in languages such as Java or C#. Object algebras do not need the most advanced and difficult features of generics available in those languages, e.g. F-bounded quantification [6], wildcards [43] or variance annotations. As a result, object algebras are applicable to a wide range of programming languages that have basic support for generics.

An important advantage of object algebras over traditional visitors is that there is no need for `accept` methods. As a consequence object algebras support *retroactive implementations* [46] of interfaces or operations without preparation of existing source code. This is unlike the Visitor pattern, which can also provide retroactive implementations only if the original classes include `accept` methods.

We discuss applications of object algebras that go beyond toy examples usually presented in solutions for the EP. In the paper we develop an increasingly more complex set of features for a mini-imperative language and describe a real-world application of object algebras in an implementation of remote batches [21, 47].

Object algebras have benefits beyond the basic extensibility of the EP. Object algebras address harder related problems, including the *expression families problem* (EFP) [30], *family polymorphism* [12] and *independent extensibility* [49].

Programming with object algebras does require learning new design strategies. Rather than create generic objects and then visit them to perform operations, object algebras encourage require that object creation be done relative to a factory, so that specialized factories can be defined to create objects with the required operations in them. Programming against factories has some cost to it because it requires parametrization of code by factories and use of generic types. However there are significant benefits in terms of flexibility and extensibility and, in comparison with other solutions to the EP using generic types [45, 3, 42, 49, 30], the additional cost is significantly smaller.

In summary, our contributions are:

- A solution to the EP using simple generic types. The solution can be used in mainstream languages like Java or C#.
- An alternative to the Visitor pattern that avoids the many of the disadvantages that pattern: it eliminates the need for accept methods; does not require preparation of the "visited" classes; and it supports extensibility.
- Various techniques for dealing with challenges that arise in realistic applications. For example, multi-sorted object algebras deal with multiple recursive types and generic combinator classes deal with independent extensibility.
- Interesting insights about the relationship between the Abstract Factory and the Visitor pattern. In some sense, factories and visitors are two faces of object algebras.

– Case study using remote batches. The Java implementation is available online.

2 Background

While there is extensive literature on the expression problem and abstract algebra in programming languages, we summarize the required background here.

2.1 The Expression Problem

Wadler's [45] formulation of the expression problem prescribes four requirements for potential solutions. Zenger and Odersky [49] add an extra requirement (independent extensibility) to that list. These requirements are summarized here:

- *Extensibility in both dimensions*: A solution must allow the addition of new data variants and new operations and support extending existing operations.
- *Strong static type safety*: A solution must prevent applying an operation to a data variant which it cannot handle using static checks.
- *No modification or duplication*: Existing code must not be modified nor duplicated.
- *Separate compilation and type-checking*: Safety checks or compilation steps must not be deferred until link or runtime.
- *Independent extensibility*: It should be possible to combine independently developed extensions so that they can be used jointly.

To illustrate the difficulty of solving the expression problem, we review two standard forms of extensibility in object-oriented languages and show how they fail to solve the problem.

Figure 1 shows an attempt to solve the EP using polymorphism. The basic idea is to define an interface `Exp` for expressions with an evaluation operation in it, and then define concrete implementations (data variants) of that interface for particular types of expressions. Note that evaluation returns a value of type `Value`. For the purposes of this paper we assume the following definitions of `Value` and its subclasses:

```
interface Value {
  Integer getInt();
  Boolean getBool();
}
class VInt implements Value {...}
class VBool implements Value {...}
```

It is easy to add new data variants to the code in Figure 1, but adding new operations is hard. For example, supporting pretty printing requires modifying the `Exp` interface and it's implementations to add a new method. However this violates "no modification" requirement. While inheritance can be used to add new

```
interface Exp {
 Value eval();
}
class Lit implements Exp {
 int x;
 public Lit(int x) { this.x = x; }

 public Value eval() {
  return new VInt(x);
}}
class Add implements Exp {
 Exp l, r;
 public Add(Exp l, Exp r) { this.l = l; this.r = r; }

 public Value eval() {
  return new VInt(l.eval().getInt() + r.eval().getInt());
}}
```

Fig. 1. An object-oriented encoding of integer expressions.

operations, the changes must be made to the interface and all classes simultaneously, to ensure static type safety. Doing so is possible, but requires advanced typing features.

An alternative attempt uses the VISITOR pattern [13]. The VISITOR pattern makes adding new operations easy, although a different interface for expressions is required:

```
interface Exp {
 <A> A accept(IntAlg<A> vis);
}
```

The `IntAlg` visitor interface, defined in Figure 2, has a (visit) method for each concrete implementation of `Exp`. These visit methods are used in the definitions of the `accept` methods. For example, the definition of the `Add` class would be:

```
class Add implements Exp {
 Exp left, right;
 public Add(Exp left, Exp right) { this.left = left; this.right = right; }

 public <A> A accept(IntAlg<A> vis) {
  return vis.add(left.accept(vis), right.accept(vis));
}}
```

There are several kinds of visitors in the literature [34]. We use a (functional) *internal visitor* [5, 34] for our example since this type of visitors will be important later in Section 5.1. An internal visitor is a visitor that produces a value by processing the nodes of a composite structure, where the control flow is controlled by the infrastructure rather than the visitor itself.

With visitors, new operations are defined in concrete implementations of visitor interfaces like `IntAlg`. Figure 5 shows a concrete visitor for pretty printing. Unlike the first solution, adding new operations can be done without modifying

```
interface IntAlg<A> {
 A lit(int x);
 A add(A e1, A e2);
}
```

Fig. 2. Visitor interface for arithmetic expressions (also a signature interface).

`Exp` and its implementations. This is especially important when dealing with objects created by library classes, because it is often impossible to change the code of the library. From a software engineering viewpoint, Visitors localize code for operations in one place, while conventional OO designs scatter code for operations across multiple classes. Visitors also provide a nice way to have state that is local to an operation (rather than to a class).

Unfortunately, traditional visitors trade one type of extensibility for another: adding new data variants is hard with visitors. The problem is the concrete references to visitor interfaces `IntAlg` in the `accept` method. Adding new data variants requires modifying `IntAlg` and all its implementations with new visit methods to deal with the new variants. Another drawback of visitors is that some initial preparation is required: the visited classes need to provide an `accept` method. This can be a problem when the source code of the classes that we want to visit is not available: if the classes have no `accept` method it is impossible to use the visitor pattern.

2.2 Algebraic Signatures, F-Algebras, and Church Encodings

An *algebraic signature* Σ [17] defines the names and types of functions that operate over one or more abstract types, called *sorts*. We assume the existence of some primitive built-in sorts for integers and booleans.

signature E
  lit: Int $\rightarrow$ E
  add: E $\times$ E $\rightarrow$ E

A general algebraic signature can contain *constructors* that return values of the abstract set, as well as *observations* that return other kinds of values. In this paper we restrict signatures to only contain constructors, as in the example given above. We call such signatures *constructive*.

An Σ-*algebra* is a set together with a collection of functions whose type is specified in the signature Σ. A given signature can have many algebras. For example, one valid E-algebra has a set of two values and simple constant operations: (E=$\{x, y\}$, lit=$\lambda n.x$, add=$\lambda(a, b).x$), where x, y are arbitrary constants. This algebra seems unsatisfying because it is degenerate, in that it ignores the inputs of its functions, and messy, in that it's set includes extra values that are never used. A special algebra, called the *initial* or *free* algebra, is neither messy nor degenerate. One way to create the initial algebra is to use a set that contains expressions, which are applications of functions in all legal ways according to the signature, and to define the functions simply as constructors. The initial algebra looks like this:

E = { lit(0), lit(1), ..., add(lit(0), lit(0)), add(lit(0), lit(1)), ... }
lit = $\lambda n.\texttt{lit}(n)$
add = $\lambda(a, b).\texttt{add}(a,\ b)$

The concept of a constructive signature defined above is a syntactic characterization of a class of algebras. A more fundamental approach comes from merging the signature's constructor functions $f_1 : T_1 \to A, ..., f_n : T_n \to A$ into a single function $f : F(A) \to A$ where F is a functor given by $F(A) = T_1 + ... + T_n$. This transformation is based on the isomorphism $(S+T) \to A \approx (S \to A) \times (T \to A)$. The function $f : F(A) \to A$ is called an F-algebra. When F is a functor built of sums and products, it can be used to give a (categorical) semantics to algebraic datatypes [25]. For example, the functor for integer expressions is $F(E) = \text{Int} + (E \times E)$. The free algebra is then the initial algebra in the category of F-algebras. Because F-Algebras provide a nice framework to formalize and reason about algebraic datatypes, they have been widely explored by the functional programming community.

It is also possible to define free algebras in a complete different way, by using Church encoding. Church encoding involves converting the algebra signature into a particular kind of polymorphic type. For example, given a signature Σ with with sort A and functions $f_1 : T_1 \to A, ..., f_n : T_n \to A$, the Church encoding is given by the type

$\text{Church}_\Sigma = \forall A.(T_1 \to A) \times ... \times (T_n \to A) \to A$

A Church encoding works by taking an algebra (sort and functions) as input and using it to create an element of the sort. Thus a Church "value" is not really a value, but rather a *recipe* for creating a value. The recipes in a Church encoding are isomorphic to the free algebra because of parametericity [15]. As a concrete example, the signature E defined above has the Church encoding:

$\text{Church}_E = \forall E.(Int \to E) \times (E \times E \to E) \to E$

When interpreted in object-oriented programming, Church encodings correspond to internal visitors [5, 34]. From a functional programming point of view, Church encodings represent data as folds [15].

3 Object Algebras

Algebraic signatures can be defined in statically typed object-oriented languages by creating a generic interface whose parameter is the abstract type. An example algebraic signature representing the abstract syntax of simple expressions is given in Figure 2, which was previously introduced as the type of an internal visitor. Interfaces representing algebraic signatures correspond closely to Abstract Factory interfaces [13]. The difference is that a factory interface typically uses a specific concrete class or interface as the result type for the factory methods, while the algebraic signature interface has an generic type. The factory interface can be derived by instantiating the abstract type to the specific object interface of the objects being created.

An *object algebra* is a class the implements an interface representing an algebraic signature. Figure 3 defines an object algebra that plays the role of a

```
class IntFactory implements IntAlg<Exp> {
 public Exp lit(int x) {
  return new Lit(x);
 }
 public Exp add(Exp e1, Exp e2) {
  return new Add(e1, e2);
}}
```

Fig. 3. Using an algebra as a factory.

factory for expressions. The factory defines how to create each kind of object in the composite structure.

To create an actual object, some part of the code will instantiate the factory and then invoke its methods repeatedly to create a specific instance. This object construction process may also be *parameterized* by the factory itself, allowing the process to create specific objects using different factories. The result is similar to a Church encoded value. For example, a function to create an expression object, and an example test function that uses it, are given below.

```
 <A> A make3Plus5(ExpAlg<A> f) {
  return f.add( f.lit(3), f.lit(5) );
 }
 void test() {
  Exp e = make3Plus5(new ExpFactory());
 }
```

A similar function could be written to parse expressions or load them from a binary representation. In these cases the use of object algebras is closer to the Builder pattern. The Builder pattern, like the Abstract Factory pattern, is a creational pattern. The main difference between the Builder pattern and the Abstract Factory pattern is that builders tend to have complex object construction processes. For example factories are typically stateless, while builders can maintain a state. In Section 6.2 we will show an example where the object construction process maintains a local state.

4 Retroactive Interface Implementations

This section shows one of the key advantages of object algebras: support for retroactive interface implementations without requiring initial preparation of code.

To illustrate retroactive implementations consider the simple object-oriented implementation of arithmetic expressions in Figure 1. These expressions support evaluation, but not pretty printing. Suppose that we now wanted to support pretty printing. Normally, as discussed in Section 2.1, we would either:

1. change the definition of the interface `Exp` to support a printing operation and change all the implementors of that interface to implement the operation; or

```
interface IPrint {
 String print();
}
class IntPrint implements IntAlg<IPrint> {
 public IPrint lit(final int x) {
  return new IPrint() {
   public String print() {
    return new Integer(x).toString();
  }};
 }
 public IPrint add(final IPrint e1, final IPrint e2) {
  return new IPrint() {
   public String print() {
    return e1.print() + " + " + e2.print();
  }};
}}
```

Fig. 4. A retroactive implementation of printing for arithmetic expressions.

2. use the VISITOR pattern, which would also require modifications in the class hierarchy to introduce `accept` methods.

Both options require pervasive changes to existing code. Furthermore, these changes are only an option if the source code is available. If the hierarchy is part of a library or framework, then these solutions are not options.

What we would like is a mechanism that allowed us to retroactively implement interfaces for existing class hierarchies, without need for changes to existing implementations.

As it turns out object algebras enable us to simulate such retroactive implementations of interfaces. To use object algebras to provide retroactive implementations we proceed very much like an implementation of internal visitors. The idea is illustrated in Figure 4. To provide the retroactive implementation of the interface, we create an implementation of the object algebra with the abstract type instantiated to the interface type. In this case `IPrint` is the interface that the arithmetic expressions should implement. The interface implementations are done by creating a class `Print` that implements the object algebra interface `IntAlg<IPrint>`. The implementations of the two methods `lit` and `add` provide the implementation of the interface for literals and addition.

The difference to the VISITOR pattern is that we do not add `accept` methods to `Exp`. Instead, following the approach presented in Section 3, we replace uses of concrete constructors in the client code by the corresponding methods in the object algebra. For example, instead of creating an expression

```
Exp exp = new Add(new Lit(3), new Lit(4))
```

we would abstract uses of the constructors as follows:

```
<A> A exp(ExpAlg<A> v) {
 return v.add(v.lit(3), v.lit(4));
}
```

With this transformation in place we could then write the following code:

```
void test() {
 Factory base = new Factory();
 Print print = new Print();

 int x = exp(base).eval(); // int x = exp.eval();
 String s = exp(print).print();
}
```

Compared to the conventional object-oriented style, uses of `exp.m()` are replaced by `exp(mFactory).m()`, where `mFactory` is a factory that creates objects with the required `m` method.

By using this simple pattern we can provide retroactive implementations of interfaces to existing code. In comparison to Java extensions such as JavaGI [46], which provide native language support for retroactive implementations, there is of course some overhead in terms of additional code. On the other hand, no new compiler is needed. One difficulty of using this pattern arises when the operations in the retroactive interface implementations depend on existing operations in the base classes or other retroactive implementations. The simple pattern presented in this section is insufficient to allow such dependencies. However, with a bit more work, we can get around this restriction as we shall see in Section 7.3.

Finally, note that this style of retroactive implementations is quite powerful: it still allows us to simulate *dynamically* dispatched methods and open classes [9]. This is unlike approaches such as C# extension methods [1] or conventional object-oriented encodings of type classes [33] which can only provide *static* dispatching in their retroactive implementations. Although the programming style required by object algebras (and retroactive interface implementations) is similar to the use of object-oriented encodings of type classes, the key difference is that object algebras overload *constructors* instead of methods.

5 Extensibility

There are two ways in which we may want to extend our expressions: adding new variants; or adding new operations. The previous section has already shown one way in which we can add new operations: via retroactive interface implementations. In this section we show another alternative way to define operations and illustrate the addition of new variants. We also show how object algebras go beyond many solutions to the EP and also provide solution to the *expression families problem* [30].

5.1 Internal Visitors as Object Algebras

Object algebras provide a direct implementation of (functional) internal visitors [34] (see also Section 2.1) since constructive algebraic signatures correspond exactly to internal visitor interfaces. As such we can use object algebras to define new operations using concrete internal visitor implementations. As Figure 5

```
class Print2 implements IntAlg<String> {
  public String lit(int x) {
    return new Integer(x).toString();
  }
  public String add(String e1, String e2) {
    return e1 + e2;
}}
```

Fig. 5. Adding a printing operation.

```
interface IntBoolAlg<A> extends IntAlg<A> {
  A bool(Boolean b);
  A iff(A e1, A e2, A e3);
}
```

Fig. 6. Adding boolean expression variants.

shows this offers an alternative way to implement pretty printing. Instead of creating a new interface like `IPrint` and defining a retroactive implementation for that type, we can directly define the printing operation. In this case it is the `IntAlg` interface that is interpreted as an internal visitor interface.

Printing is used as follows:

```
Print2 p = new Print2();
String s = exp(p);
```

This object algebra visitor style avoids the creation of an intermediate object, just to immediately invoke the `print` method afterwards. Unlike traditional visitor implementations, this visitor style using object algebras supports data variant extensibility and does not need `accept` methods.

Using internal visitors is best when the computation in the operation happens bottom-up: essentially operations that could be defined as folds in functional programming. This steams, of course, from the fact that internal visitors are basically Church encodings and Church encodings encode data as folds. For operations that do not naturally fit this bottom-up style of computation, or mutually depend on other operations, the factory-oriented approach using retroactive implementations of interfaces is better.

5.2 Adding New Variants and Updating Operations

Adding new data variants is easy. The first step is to create new classes `Bool` and `Iff` in the usual object-oriented style (like `Lit` and `Add`):

```
class Bool implements Exp {...}
class Iff implements Exp {...}
```

The second step, shown in Figure 6, is to create an extended algebra interface with two new methods for the new boolean expressions. Finally the last step, shown in Figure 7, is to provide extension for the new boolean expressions cases for both the factory `Factory` and the retroactive implementation for printing.

```
/* Extended Expression Factory */
class IntBoolFactory extends IntFactory implements IntBoolAlg<Exp> {
 public Exp bool(Boolean b) {
  return new Bool(b);
 }
 public Exp iff(Exp e1, Exp e2, Exp e3) {
  return new Iff(e1, e2, e3);
}}
/* Extended Retroactive Implementation for Printing */
class IntBoolPrint extends IntPrint implements IntBoolAlg<IPrint> {
 public IPrint bool(final Boolean b) {
  return new IPrint() {
   public String print() {
    return new Boolean(b).toString();
   }};
 }
 public IPrint iff(final IPrint e1, final IPrint e2, final IPrint e3) {
  return new IPrint() {
   public String print() {
    return "if (" + e1.print() + ") then " + e2.print() + " else " + e3.
        print();
  }};
}}
```

Fig. 7. Supporting boolean expression variants.

5.3 Subtyping Relations

There are two interesting subtyping relations when we talk about the EP: subtyping between extended and base terms; and subtyping between the operations on those terms. Object algebras support both types of subtyping.

Not many other solutions to the EP support such subtyping relations. Even using advanced features like virtual classes [24] and similar mechanisms [4, 26, 28], the extended terms and operations are incompatible with the base terms and operations: only subtyping relations between classes in the *same family* are preserved. Oliveira [30] recognized this problem and suggested a variant of the EP: the *expression families problem*, which requires solutions to preserve the subtyping relations *across different families*. There are two solutions that we are aware of that do support such subtyping relations [42, 30]. Still both of them require variance annotations or wildcards.

Subtyping between object algebra interfaces follows from standard OO subtyping: an extension of an object algebra interface is a *subtype* of the original interface. A consequence of this subtyping relation is that if we have some term constructed using a certain object algebra, we can always use an operation defined over an extension of that object algebra to process the term. For example:

```
IntBoolPrint p2 = new IntBoolPrint();
exp(p2).print();
```

In this case we can use `p2` (which supports integer and boolean expressions) in an integer expression. However, the following code would be rejected:

```
IntPrint p = new IntPrint();
exp2(p).print(); // type-error
```

Here, we create a printing implementation `p` for integer expressions and try to use it on an expression `exp2` (see the definition below) defined for integer and boolean expressions. As expected, this fails to type-check.

The subtyping relation between terms is induced by the subtyping relation between object algebra interfaces. However, it follows the opposite direction: an extended term type (that is, with more constructors) is a *supertype* of a base term type. This subtyping relation is useful, for example, to build complex terms using an extended set of constructors from simpler terms:

```
<A> A exp(IntAlg<A> v) {
 return v.add(v.lit(3), v.lit(4));
}
<A> A exp2(IntBoolAlg<A> v) {
 return v.iff(v.bool(false),exp(v),v.lit(0));
}
```

In this case `exp` is a type of terms which can only be built using integer expressions, whereas `exp2` is type of terms which can use boolean expressions as well. Since terms for integer expressions are a subtype of terms for integer and boolean expressions, we can use `exp` in the construction of `exp2`.

Finally, note that there is an important difference to solutions to the EP using open classes [9]. In those solutions, there is a *single* expression type which is incrementally extended. So, once expressions are extended it becomes impossible to distinguish the extended expressions from more basic expressions. Thus, unlike a solution with object algebras, type distinctions between multiple variations of expressions are lost.

6 Multiple Types and Multi-sorted Object Algebras

In larger programs, it is often the case that we need multiple (potentially mutually) recursive types and operations evolving as a family. When the need for multiple types arises, we need to generalize from simple object algebras to multi-sorted object algebras. Multi-sorted object algebras also illustrate the relationship with the Abstract Factory pattern better, since this pattern is normally used with complex hierarchies with multiple types.

Multi-sorted object algebras are closely related to *family polymorphism* [12] which allow a variation EP where multiple types evolve as a family. Normally, without mechanisms like virtual types or classes, family polymorphism tends to be extremely heavyweight (and impractical) to encode [39]. However, as we shall see object algebras still scale well to the multiple type case.

6.1 Multiple Types

The need for multiple types appear, for example, when we want to have a language with expressions and statements. Figure 8 shows how to add statements

```
interface StmtAlg<E, S> extends IntBoolAlg<E> {
 E var(String x);
 E assign(String x, E e);
 S expr(E e);
 S comp(S e1, S e2);
}
```

Fig. 8. Statements and expressions as a multi-sorted object algebra.

to our little language of boolean and integers expressions. In order to introduce a new syntactic sort (statements) in the language, we need to add a new type parameter `S`. This corresponds effectively to having a multi-sorted (object) algebra, with `E` and `S` as the carrier types. [1]

As part of the statements object algebra interface we introduce two new forms of expressions: variables (`var`) and assignments (`assign`). We also introduce two forms of statements: sequential composition (`comp`) and liftings of expressions into statements (`expr`).

6.2 Evaluation of statements: Algebras with Local State

Evaluation of statements is interesting for two reasons. Firstly it illustrates the definition of multi-sorted object algebras. Secondly it also illustrates an operation with local state: namely, the mapping between variables and values associated with those variables. If we would design the evaluation of statements using a more conventional OO style, with independent classes for variables and assignments, then we would have to coordinate the mapping of variables between those two classes. This could be done, for example, by explicit passing the variable mapping between those two classes. However, because this mapping is basically local to evaluation, this design is a bit unfortunate as it loses some encapsulation.

With the Visitor pattern we could solve this problem more elegantly because we could create state that is local to an operation. Because object algebras also offer some of the same benefits as visitors, we can also exploit this design in our case. To do so, we use a design that is similar to retroactive interface implementations. This design is illustrated in Figure 9. Instead of creating individual classes, we use inner anonymous classes directly in the factory. The variable `map` keeps the mapping between variables and values. In the case of variables, we use `map` to retrieve the value associated with that variable. Assignments update the variable in the map with the value of the assigned expression. Composition evaluates the two expressions sequentially. Finally, `expr` simply returns the corresponding expression.

Client code: The extended object algebra of expressions and statements can be used as before. For example we can create values for expressions and statements as follows:

[1] Note that, more generally, we can encode sets of n (potentially mutually) recursive types by creating multi-sorted algebras with n type parameters (one for each type).

```
interface Stmt {
 void eval();
}
class StmtFactory extends IntBoolFactory implements StmtAlg<Exp, Stmt> {
 HashMap<String, Value> map = new HashMap<String, Value>();

 public Exp var(final String x) {
  return new Exp() {
   public Value eval() {
    return map.get(x);
   }};
 }
 public Exp assign(final String x, final Exp e) {
  return new Exp() {
   public Value eval() {
    Value v = e.eval();
    map.put(x, v);
    return v;
  }};
 }
 public Stmt comp(final Stmt s1, final Stmt s2) {
  return new Stmt() {
   public void eval() {
    s1.eval();
    s2.eval();
  }};
 }
 public Stmt expr(final Exp e) {
  return new Stmt() {
   public void eval() {
    e.eval();
  }};
}}
```

Fig. 9. An abstract factory for expressions and statements.

```
<E,S> E exp(StmtAlg<E,S> v) {
 return v.assign("x", v.add(v.lit(3), v.lit(4)));
}
<E,S> S stmt(StmtAlg<E,S> v) {
 return v.comp(v.expr(exp(v)), v.expr(v.var("x")));
}
```

Note that the syntactic restrictions which dictate where a expressions and statements can occur are preserved. As such code like:

```
<E,S> S badStmt(StmtAlg<E,S> v) {
 return v.comp(exp(v), v.var("x")); //type-error
}
```

is rejected by the type-checker since it tries to use two expressions as arguments for sequential composition.

```
interface BoolAlg<A> {
  A bool(boolean x);
  A iff(A b, A e1, A e2);
}

interface ExpIntBool<A> extends BoolAlg<A>, IntAlg<A> {}
```

Fig. 10. Composing algebra interfaces with interface inheritance.

The evaluator is run as before: a factory is created, passed to `exp` and `stmt` and `eval` is invoked in the resulting objects.

```
StmtFactory factory = new StmtFactory();
exp(factory).eval();
stmt(factory).eval();
```

7 Modularity and Object Algebra Combinators

This section shows techniques to modularly define and compose independent components using object algebra combinator classes. One of the problems addressed in this section is how to achieve *independent extensibility* [49] in Java (Section 7.2). It is easy to have independent extensibility if a language supports traits [40] or mixin composition [2], but this is not as trivial in a language with single inheritance like Java. Another problem that is addressed in this section is the problem of defining retroactive implementations that depend on existing operations in the base classes or other retroactive implementations (Section 7.3).

7.1 Modular Combination of Algebra Interfaces

Interface inheritance can be used to combine algebra interfaces. For example, lets consider again the problem of developing boolean and integer expressions. In Figure 6 we opted to make the algebra interface for boolean expression extend that of integer expressions. However, there's nothing intrinsic to boolean expressions that depends on integer expressions. A more modular alternative implementation, shown in Figure 10, is to define integer and boolean algebras as separate interfaces. Because most languages support multiple interface inheritance, this mechanism can compose the two algebra interfaces. The new interface `IntBoolAlg` illustrates this idea and shows how to compose `IntAlg` with `BoolAlg` through interface inheritance.

7.2 Modular Combination of Algebras

Unfortunately modular combinations of algebras themselves is not as easy as modular combination of algebra interfaces. The problem is that while languages

```
class Union<A> implements IntBoolAlg<A> {
 BoolAlg<A> v1;
 IntAlg<A> v2;
 Union(BoolAlg<A> v1, IntAlg<A> v2) { this.v1 = v1; this.v2 = v2; }

 public A lit(int x)    { return v2.lit(x);   }
 public A add(A e1, A e2) { return v2.add(e1, e2); }
 public A bool(Boolean b) { return v1.bool(b); }
 public A iff(A e1, A e2, A e3) { return v1.iff(e1, e2, e3); }
}
```

Fig. 11. Composing operations with OO composition.

like Java support multiple interface inheritance, they only support single implementation inheritance. As we such we cannot use implementation inheritance in Java to compose two independent extensions.

However, OO composition offers an alternative way to combine modular extensions, although it takes some manual work to set up the composition. Fortunately it is possible to write fairly generic composition classes which allow composing different types of interpretations. At the high-level what we want is to define a combinator:

$union \in V_1\ A \times V_2\ A \to (V_1 \otimes V_2)\ A$

which takes two object algebras of type $V_1\ A$ and $V_2\ A$ and it returns the union of those algebras. In Java we can write *union* for two specific object algebras interfaces $V_1\ A$ and $V_2\ A$. Figure 11 illustrates the definition of *union* for the object algebras interfaces `BoolAlg` and `IntAlg`. We can use Java's multiple interface inheritance to approximate the union of two object algebras interfaces (that is the type-level operator $\otimes$). The actual implementation of the methods of `Union` is straightforward: each method simply delegates to the corresponding method in either `v1` or `v2`.

`Union` can be used to define the factory for boolean and arithmetic expressions from two independent extensions:

```
class IntBoolFactory2 extends Union<Exp> {
 IntBoolFactory2() { super(new BoolFactory(), new IntFactory()); }
}
```

Essentially all we have to do is to instantiate factories for boolean and integer expressions and invoke the constructor in `Union`. For retroactive implementations such as pretty printing we would proceed in the same way.

7.3 Combining Operations in Parallel

Sometimes it is useful to compose multiple operations together in such a way that they are executed in parallel to the same input. Abstractly speaking what we want is a combinator:

$combine \in VA \times VB \to V(A \times B)$

```
class Pair<A, B> {
 A a; B b;
 Pair(A a, B b) { this.a = a; this.b = b; }

 A a() { return a; }
 B b() { return b; }
}

class Combine<A, B> implements IntAlg<Pair<A, B>> {
 IntAlg<A> v1;
 IntAlg<B> v2;

 Combine(IntAlg<A> v1, IntAlg<B> v2) { this.v1 = v1;this.v2 = v2; }

 public Pair<A, B> lit(int x) {
  return new Pair<A, B>(v1.lit(x), v2.lit(x));
 }
 public Pair<A, B> add(Pair<A, B> e1, Pair<A, B> e2) {
  return new Pair<A, B>(v1.add(e1.a(), e2.a()), v2.add(e1.b(), e2.b()));
 }
}
```

Fig. 12. Combining operations in parallel.

That is given two object algebras with types VA and VB we want to derive a third object algebra which combines the results of the two object algebras. This combinator is analogous to the `zip` function in functional programming, and it has been well-studied in the context of F-algebras [20].

Figure 12 shows how to define `combine` for integer expressions. Essentially, *combine* becomes an class (`Combine`) that is parametrized by two other object algebras `v1` and `v2`. The implementation of each method (`lit` and `add`) basically forwards the input to the corresponding cases in `v1` and `v2` and returns a pair with both results.

`Combine` is useful, for example, when we need to define operations that depend on multiple independent extensions. For example, consider adding some debugging information to the evaluator. In order to do this it is helpful to have a pretty printer. However, evaluation and pretty printing have been defined separately. By inheriting from `Combine` we can create a new class `Debug` that allows us to use evaluation and pretty printing at the same time.

```
class Debug extends Combine<Exp, IPrint> {
 Debug() { super(new IntFactory(), new IntPrint()); }
 Pair<Exp, IPrint> add(Pair<Exp, IPrint> e1, Pair<Exp, IPrint> e2) {
  System.out.println("The first expression " + e1.snd().print() +
     " evaluates to " + e1.fst().eval().toString());
  System.out.println("The second expression " + e2.snd().print() +
     " evaluates to " + e2.fst().eval().toString());
  return super.add(e1,e2);
}}
```

all we have to do is to invoke the constructor of the super class (`Combine`) with the integer expressions factory and pretty printer. Then to access the pretty printer and the evaluator, we just select the right component of the combined pair.

7.4 Some Final Notes on Extensibility

The attentive reader may have noticed two additional extensibility challenges that were left unaddressed. The first challenge is that the algebra class combinators that were just introduced are not extensible. The second challenge is that while expressions (and statements) are extensible the values computed by evaluation are not.

Both problems can be solved, but they require the most advanced use of generics in this paper: bounded polymorphism. However, we are still able to avoid F-bounded polymorphism since there is no need for recursive F-bounds.

We describe the key ideas of the solutions next, but refer the reader to our online implementation for the full code.

Extensible algebra combinators: Both `Union` and `Combine` cannot be easily extended. This is because these classes require concrete classes like `IntAlg` or `BoolAlg` in order to refer to the types of the object algebra parameters. It would be quite unfortunate if those algebra combinator classes could not be extended, because this would mean that each extension would have to create new algebra combinator classes from scratch.

To make such algebra combinator classes extensible we first observe that in `Union` and `Combine` there is no need to know about the concrete classes of the object algebra parameters. Rather only the upper bounds matter. Exploiting this observation we can define generalized versions of `Union` and `Combine` as follows:

```
class GUnion<A, V1 extends BoolAlg<A>, V2 extends IntAlg<A>> implements
    IntBoolAlg<A> {
 V1 v1; V2 v2;
 GUnion(V1 v1, V2 v2) { this.v1 = v1; this.v2 = v2; }
 ...
}
class GCombine<A, B, V1 extends IntAlg<A>, V2 extends IntAlg<B>>
       implements IntAlg<Pair<A, B>> {
 V1 v1; V2 v2;
 GCombine(V1 v1, V2 v2) { this.v1 = v1;this.v2 = v2; }
 ...
}
```

Unlike `Union` and `Combine` these classes allow extensibility because the bounds can be refined when the classes are extended.

Extensible values: A similar idea is used to allow extensible values as well as extensible expressions.

```
interface BatchFactory<E> {
 E Var(String name); // variable reference
 E Data(Object value); // simple constant (number, string, or date)
 E Fun(String var, E body);
 E Prim(Op op, List<E> args); // unary and binary operators
 E Prop(E base, String field); // field access
 E Assign(Op op, E target, E source); // assignment
 E Let(String var, E expression, E body); // control flow
 E If(E condition, E thenExp, E elseExp);
 E Loop(Op op, String var, E collection, E body);
 E Call(E target, String method, List<E> args); // method invocation
 E In(String location); // reading and writing forest
 E Out(String location, E expression);
}
```

Fig. 13. Batch script language abstract syntax

```
interface IntVal<A> {
 A lit(int x);
}
interface IntExp<A> extends IntVal<A> {
 A add(A x, A y);
}
interface IntValue {
 int getInt();
}
class Eval<A extends IntValue,V extends IntVal<A>> implements IntExp<A> {
 protected V valFact;
 public Eval(V valFact) { this.valFact = valFact; }
 public A lit(int x) { return valFact.lit(x); }
 public A add(A x, A y) { return valFact.lit(x.getInt() + y.getInt()); }
}
```

Integer value factories are described by the `IntVal` algebra interface. There's a single constructor `lit`. The algebra interface for expressions then becomes an extension of `IntVal`. We also need an integer value interface (`IntValue`) for evaluation. With these 3 interfaces we can define an evaluation class (`Eval`) by parametrizing that class by a value factory. This factory is then used to avoid the use of concrete value constructors (as done in Figure 1).

8 Case Study

We have used this technique in implementing a new client model for invoking remote procedure calls (RCP), web services, and database clients (SQL). The system is called *ibrahim09remote,Wiedermann11*. The system uses a custom scripting language to communicate batches of operations from clients to servers. The base object algebra of the system is defined in Figure 13. Some helper functions (which are technically redundant) are omitted for brevity.

```
interface SQLTranslation {
 void toSQL(StringBuilder sb, List<Object> params, Forest data);
 Expression normalize(ISchema schema, SQLQuery query,
    Expression outerCond, Env env, NormType normType);
 SQLTable getTable();
 Expression invertPath(Expression e, Env env, boolean fromChild);
 SQLTable getTableNoJoins(Env env);
 SQLTable getBase(Env env);
 Expression withoutTransformations();
 Expression getTransformations(Expression base);
 Expression trimLast(Env env);
}
```

Fig. 14. Interface of mutually recursive methods used by the SQL Translation algebra

```
interface PartitionFactory<E> extends BatchFactory<E> {
 E Other(Object external, E... subs);
 E DynamicCall(E target, String method, List<E> args);
 E Mobile(String type, Object obj, E exp);
}
```

Fig. 15. Extended script language interface used for partitioning algebra

There are currently five implementations of this signature. The first three implement direct evaluation of scripts, secure evaluation, and SQL translation, respectively.

The direct evaluation classes are similar to the classes defined in Section 4. The secure evaluation classes have the same basic structure, but carry additional state so that they can check the legality of each operation for the current user.

The SQL translator injects the algebra into a object hierarchy that has multiple mutually recursive translation functions, and also has additional SQL-specific objects. The signature of this class is given in Figure 14. Using a traditional visitor approach, every one of these functions would have to be defined as a mutually recursive visitor class. With object-algebras, the SQL translation objects can call methods on sub-objects in the normal object-oriented style.

The final implementations are the most complex. It implements the partitioning mechanism required by the batch compiler. There are two parts, a partitioning algebra and a code generation algebra. The partition algebra extends the base algebra signature with additional node types, to represent code that does not belong to the batch. The methods, which are also mutually recursive, are listed in Figure 15. The partition system then creates batches, but then must *visit* the resulting objects to generate new code after the partition is complete. This is the one case where something like a traditional visitor is used. However, it is not used to create new operations. Instead it is used to *build* the new objects into a final code-generation algebra.

Subjectively this architecture allows the different subsystems (security, partitioning, and SQL translation) to be kept separate. Within each subsystem

ordinary object-oriented dispatch is used. The main difference is that rather than constructing generic operations, and then trying to create complex mutually recursive visitors that operation on the generic objects, the batch system creates specialized objects for each task.

9 Related Work

Throughout the paper we have already compared object algebras with several other related work. In this section we discuss additional related work.

Expression Problem in Java-like languages: Object algebras require only simple generics and work in languages like Java or C#. As far as we know Torgersen's [42] work on the EP presents the only solutions in the literature that can also work in those languages. He presents 4 solutions, but the first 3 solutions require advanced features like F-bounds or wildcards, while the last solution makes use of C# reflection mechanisms and it does not satisfy the (static) type-safety requirement of the expression problem. A drawback of Torgersen first 3 solutions is that they require quite a bit of redundant code just for the purposes of satisfying the type-checker, and they are conceptually quite heavy. F-bounds and wildcards are notorious for being difficult to grasp for everyday programmers. An advantage of object algebras is that they are comparably lightweight on the amount of type annotations and they do not use those advanced features.

Before Torgersen's work, there have been attempts to provide solutions that work in Java-like languages. Wadler proposed a solution using generics to solve the expression problem [45], but he later found a subtle typing problem. Kim Bruce [3] proposed a solution to the expression problem using generics and *self-types*. However self-types are not a widely available feature and, as such, his solution does not work in most current mainstream OO languages. Finally, there have also been some other solutions that are not statically type-safe, but still allow extensibility [22, 35, 44].

Modular visitors, encodings of datatypes and embedded DSLs: There has been a considerable amount of work on modular visitors and related techniques in advanced programming languages like Haskell or Scala recently. This line of work is closely related to object algebras.

Hinze [18] was the first to point out that type classes provide a way to represent encodings of datatypes in Haskell. He exploited this fact to implement a generic programming library. Inspired by Hinze's work, Oliveira and Gibbons [31] have shown general patterns for those techniques and used them to several other applications. In following work Oliveira et al. showed that variants of these type-class based encodings are extensible and can be used to solve the expression problem [32]. Later work by Carrete et al. [7] and Hofer et al. [19] (in Scala) popularized those techniques for defining well-typed interpreters and embedded DSLs. While all that work is closely related to object algebras, those techniques require significant advanced language features not available in mainstream OO languages like Java. A source of additional complexity in this line

of work is that most documented applications use encodings of *generalized* algebraic datatypes [36]. Even with our simplified techniques, it is not possible to define Church encodings of generalized algebraic datatypes in Java as this requires type-constructor polymorphism [38, 27].

The relationship between the VISITOR pattern and Church encodings has been folklore in the type-theory community. Buchlovsky and Thielecke [5] were the first to precisely document that relationship. The link between the type class based encodings, the VISITOR pattern, encodings of datatypes and the extensibility of such encodings was further developed by Oliveira [29, 34]. In later work, Oliveira [30] generalized and showed how to apply his results on extensibility to two variations of the VISITOR pattern. Still the Scala implementation of that work used features like: type constructor polymorphism, variance annotations, self-type annotations and mixins. None of these are available in Java. A key insight of our work is that by avoiding `accept` methods and using plain object algebras instead we can get most of the benefits of modular visitors in Java. Furthermore, unlike visitors, object algebras support allow retroactive implementations without requiring `accept` methods.

Another Haskell solution to the expression problem is based on folds (and F-algebras) [11, 41]. This solution also requires advanced features and it does not translate well to object-oriented programming because most OO languages do not have native support for sums-of-products, which are needed in that solution.

Finally Zenger and Odersky [49] proposed a solution to the expression problem in Scala using virtual types [4] and the *open class pattern* [26]. As most other solutions discussed here, this solution is quite heavyweight in terms of language features and it requires a lot of type annotations and manual composition code.

Language-based solutions to the expression problem: Several programming language features such as *multi-methods* [8], *open classes* [9], *virtual classes* [24, 28], *virtual types* [4], *units* [26], *polymorphic variants* [14] and others [48, 23, 46] are aimed at solving problems related to the expression problem. The main advantage of most of these approaches is that solutions to the expression problem can be expressed quite naturally. In contrast, solutions that instead exploit general programming language features (like generics or type classes) are commonly criticized for being heavyweight, hard to use and require sophisticated features. Our work shows that is possible to significantly reduce such complexity by using object algebras. There is still some price to pay in terms of indirection, but compared to other approaches using general programming language features this is a relatively small cost: it is low enough that object algebras are useful in practice. The main advantage of object algebras over language-based approaches is that object algebras do not require a new language or language extension: they can be used in any languages that support a simple form of generics.

Visitor Combinators and Functional Interpretations of Design Patterns: There has been some work on visitor combinators for offering better traversal control. Visser [44] presented a number of combinators that can express interesting

traversal strategies like bottom-up, top-down or sequential composition of visitors. This work is related to our algebra combinators in Section 7. However a difference is that we use *functional* style object algebras whereas Visser uses *imperative* style visitors. As part of our future work we would like to explore more algebra combinators and develop a small algebra of combinators for object algebras.

Gibbons [16] has proposed functional interpretations for various design patterns. In particular he suggested that the VISITOR and the BUILDER patterns are closely related to folds in functional programming. Our work supports this idea and extends it, suggesting that the ABSTRACT FACTORY pattern is also also part of the functional interpretation as folds.

10 Conclusion

This paper presents a new solution to the expression problem based on object algebras. This solution is interesting because it is extremely lightweight in terms of required language features; has a low conceptual overhead for programmers; and it scales well with respect to other challenges related to the expression problem.

Object algebras promote a factory-based programming style where concrete constructors are avoided. This programming style has some overhead over a conventional object-oriented programming style, but it also offers several advantages in terms of extensibility. In comparison with the VISITOR pattern, object algebras retain most of the advantages and additionally they support extensibility and do not require `accept` methods. As such object algebras can provide retroactive implementations even when the original source code is not available.

Although this paper shows that object algebras can be encoded in languages like Java, programming language extensions are still useful. With additional language support we expect programming with object algebras to be even more convenient. For example programming language extensions can be useful to manage factories better, or to automatically provide composition operators for object algebras. This is something we would like to explore in future work.

References

1. Bierman, G.M., Meijer, E., Torgersen, M.: Lost in translation: formalizing proposed extensions to c#. In: OOPSLA '07 (2007)
2. Bracha, G., Cook, W.: Mixin-based inheritance. In: OOPSLA/ECOOP '90 (1990)
3. Bruce, K.: Some challenging typing issues in object-oriented languages: Extended abstract. Electronic Notes in Theoretical Computer Science 82(8), 1 – 29 (2003)
4. Bruce, K.B., Odersky, M., Wadler, P.: A statically safe alternative to virtual types. In: ECOOP'98 (1998)
5. Buchlovsky, P., Thielecke, H.: A type-theoretic reconstruction of the visitor pattern. Electronic Notes in Theoretical Computer Science 155(0), 309 – 329 (2006)
6. Canning, P., Cook, W., Hill, W., Olthoff, W., Mitchell, J.C.: F-bounded polymorphism for object-oriented programming. In: FPCA '89 (1989)

7. Carette, J., Kiselyov, O., Shan, C.c.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. J. Funct. Program. 19, 509–543 (September 2009)
8. Chambers, C., Leavens, G.T.: Typechecking and modules for multimethods. ACM Trans. Program. Lang. Syst. 17, 805–843 (November 1995)
9. Clifton, C., Leavens, G.T., Chambers, C., Millstein, T.: Multijava: modular open classes and symmetric multiple dispatch for java. In: OOPSLA '00 (2000)
10. Cook, W.R.: Object-oriented programming versus abstract data types. In: Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages. pp. 151–178. Springer-Verlag (1991)
11. Duponcheel, L.: Using catamorphisms, subtypes and monad transformers for writing modular functional interpreters. (1995), `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.11.7093`
12. Ernst, E.: Family polymorphism. In: ECOOP '01 (2001)
13. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns : Elements of Reusable Object-Oriented Software. Addisson-Wesley professional computing series, Addisson-Wesley (1994)
14. Garrigue, J.: Programming with polymorphic variants (1998)
15. Ghani, N., Uustalu, T., Vene, V.: Build, augment and destroy. universally. In: APLAS'04 (2004)
16. Gibbons, J.: Design patterns as higher-order datatype-generic programs. In: WGP '06 (2006)
17. Guttag, J.V., Horning, J.J.: The algebraic specification of abstract data types. Acta Informatica (1978)
18. Hinze, R.: Generics for the masses. Journal of Functional Programming 16(4-5), 451–483 (2006)
19. Hofer, C., Ostermann, K., Rendel, T., Moors, A.: Polymorphic embedding of dsls. In: GPCE '08 (2008)
20. Hoogendijk, P., Backhouse, R.: When do datatypes commute? In: Category Theory and Computer Science, 7th International Conference, volume 1290 of LNCS (1997)
21. Ibrahim, A., Jiao, Y., Tilevich, E., Cook, W.R.: Remote batch invocation for compositional object services. In: ECOOP'09 (2009)
22. Krishnamurthi, S., Felleisen, M., Friedman, D.P.: Synthesizing object-oriented and functional design to promote re-use. In: ECOOP'98 (1998)
23. Löh, A., Hinze, R.: Open data types and open functions. In: PPDP '06 (2006)
24. Madsen, O.L., Moller-Pedersen, B.: Virtual classes: a powerful mechanism in object-oriented programming. In: OOPSLA '89 (1989)
25. Malcolm, G.: Algebraic Data Types and Program Transformation. Ph.D. thesis, Rijksuniversiteit Groningen (September 1990)
26. McDirmid, S., Flatt, M., Hsieh, W.C.: Jiazzi: new-age components for old-fasioned java. In: OOPSLA '01 (2001)
27. Moors, A., Piessens, F., Odersky, M.: Generics of a higher kind. In: OOPSLA '08 (2008)
28. Nystrom, N., Qi, X., Myers, A.C.: J&: nested intersection for scalable software composition. In: OOPSLA '06 (2006)
29. Oliveira, B.C.d.S.: Genericity, extensibility and type-safety in the Visitor pattern. Ph.D. thesis, Oxford University Computing Laboratory (2007)
30. Oliveira, B.C.d.S.: Modular visitor components. In: ECOOP'09 (2009)
31. Oliveira, B.C.d.S., Gibbons, J.: Typecase: a design pattern for type-indexed functions. In: Haskell '05 (2005)

32. Oliveira, B.C.d.S., Hinze, R., Löh, A.: Extensible and modular generics for the masses. In: Trends in Functional Programming (2006)
33. Oliveira, B.C.d.S., Moors, A., Odersky, M.: Type classes as objects and implicits. In: OOPSLA '10 (2010)
34. Oliveira, B.C.d.S., Wang, M., Gibbons, J.: The visitor pattern as a reusable, generic, type-safe component. In: OOPSLA '08 (2008)
35. Palsberg, J., Jay, C.B.: The essence of the visitor pattern. In: COMPSAC'98 (1998)
36. Peyton Jones, S., Vytiniotis, D., Weirich, S., Washburn, G.: Simple unification-based type inference for gadts. In: ICFP '06 (2006)
37. Reynolds, J.C.: User-defined types and procedural data structures as complementary approaches to type abstraction. In: Schuman, S.A. (ed.) New Directions in Algorithmic Languages, pp. 157–168 (1975)
38. Reynolds, J.: Towards a theory of type structure. In: Programming Symposium, Lecture Notes in Computer Science, vol. 19, pp. 408–425 (1974)
39. Saito, C., Igarashi, A.: The essence of lightweight family polymorphism. Journal of Object Technology pp. 67–99 (2008)
40. SchÃďrli, N., Ducasse, S., Nierstrasz, O., Black, A.P.: Traits: Composable units of behaviour. In: ECOOP'03 (2003)
41. Swierstra, W.: Data types à la carte. Journal of Functional Programming 18(4), 423 – 436 (2008)
42. Torgersen, M.: The expression problem revisited – four new solutions using generics. In: ECOOP'04 (2004)
43. Torgersen, M., Hansen, C.P., Ernst, E., von der Ahé, P., Bracha, G., Gafter, N.: Adding wildcards to the java programming language. In: SAC '04 (2004)
44. Visser, J.: Visitor combination and traversal control. In: OOPSLA '01 (2001)
45. Wadler, P.: The Expression Problem. Email (Nov 1998), discussion on the Java Genericity mailing list
46. Wehr, S., Thiemann, P.: Javagi: The interaction of type classes with interfaces and inheritance. ACM Trans. Program. Lang. Syst. 33 (July 2011)
47. Wiedermann, B., Cook, W.R.: Remote batch invocation for SQL databases. In: The 13th International Symposium on Database Programming Languages (DBPL) (2011)
48. Zenger, M., Odersky, M.: Extensible algebraic datatypes with defaults. In: ICFP '01 (2001)
49. Zenger, M., Odersky, M.: Independently extensible solutions to the expression problem. In: FOOL'05 (2005)