# Extensible Language Implementation with Object Algebras (Short Paper)

Maria Gouseti
CWI, Amsterdam, The Netherlands
mgouseti@gmail.com

Chiel Peters
CWI, Amsterdam, The Netherlands
chiel.peters@student.uva.nl

Tijs van der Storm
CWI, Amsterdam, The Netherlands
storm@cwi.nl

## Abstract

Object Algebras are a recently introduced design pattern to make the implementation of recursive data types more extensible. In this short paper we report our experience in using Object Algebras in building a realistic domain-specific language (DSL) for questionnaires, called QL. This experience has led to a simple, yet powerful set of tools for the practical and flexible implementation of highly extensible languages.

***Categories and Subject Descriptors*** D.2.11 [*Software Architecture*]: Languages; D.3.2 [*Language Classifications*]: Extensible Languages; D.3.3 [*Language Constructs and Features*]: Frameworks

***General Terms*** Design, Languages

***Keywords*** Object Algebras; extensibility; interpreter

## 1. Introduction

Object Algebras are a programming technique to make the implementation of recursive data types more extensible [2]. As a solution to the "expression problem" [4], it supports modular extension of both language constructs (e.g., expressions) and operations (e.g., evaluation, type checking, etc.).

The key idea of Object Algebras is to describe the abstract syntax of a language using generic factory interfaces. Each factory method corresponds to a constructor of a data type variant. For instance, the following interface declares a data type for expressions, supporting literals and addition:

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

Operations over the data type are realized by implementing this interface and instantiating the type parameter to a concrete type. For instance, evaluation could be realized as follows[1]:

```
interface IEval { int eval(); }

class Eval implements ExpAlg<IEval> {
  IEval lit(int n) { return () -> n; }
  IEval add(IEval l, IEval r) {
    return () -> l.eval() + r.eval();
  }
}
```

The functional interface IEval captures the operation we're interested in. The class Eval acts as a factory for creating expressions that can be evaluated.

Operations for particular expressions are created by calling factory methods on implementations of the generic interface. For instance, the following generic method creates the expression "1 + 2" over a specific algebra alg:

```
<X> X make(ExpAlg<X> alg) {
  return alg.add(alg.lit(1), alg.lit(2));
}
```

To evaluate expressions, one would call this method with an instance of Eval.

Implementing a different operation involves implementing the generic interface again. This solves half of the expression problem: extensibility of operations. The other half, extension of variants, is solved by first extending the generic interface, for instance, MulAlg<E> **extends** ExpAlg<E>). Operations are then extended by implementing the extended interface and subclassing the class representing the base operation. For instance, EvalMul **extends** Eval **implements** MulAlg<IEval>.

A consequence of the object algebra style in the context of language implementation is that the notion of an AST in a sense disappears. Instead, each operation requires the (re)construction of the program structure over the algebra that performs that operation.

Object Algebras were invented only recently (2012) and have never been applied in a realistic language implementation project. In this short paper we share our experience in using Object Algebras in the implementation of a DSL for questionnaires, called QL. In particular, we address the following problems:

- In practical language implementation, program structures are created by a parser. To prevent parsing multiple times for each operation (e.g., type checking, evaluation etc.) we present a recorder combinator which delays the instantiation of a program into a concrete algebras till after parsing
- In our experience, the use of inheritance tends to lead to inflexible designs. We present a dynamic union combinator to combine languages without the use of inheritance.
- Although Object Algebras support extension of abstract syntax, they do not solve the problem of extensible concrete syntax. We

[1] We will use Java 8 features such as functional interfaces and closure literals throughout.



*GPCE'14*, September 15-16, 2014, Västerås, Sweden
Copyright 2014 ACM 978-1-4503-3161-6/14/09...$15.00
http://dx.doi.org/10.1145/2658761.2658765

```
static <T> T recorder(Class<T> ialg) {
  return (T)Proxy.newProxyInstance(ialg.getClassLoader(),
            new Class<?>[]{ialg},
            (x, m, args) -> new Builder(m, args));
}

class Builder {
  Method method; Object[] args;
  <T> T build(Object alg) {
    Object[] builtArgs = new Object[args.length];
    for (int i = 0; i < args.length; i++)
      builtArgs[i] = args[i] instanceof Builder
                   ? args[i].build(alg) : args[i];
    return (T)method.invoke(alg, builtArgs);
  }
}
```

**Figure 1.** Recording factory method calls to delay instantiation

provide a practical solution to this problem by annotating factory methods with concrete syntax productions, and generating the parsers automatically.

The combinators and patterns have been applied in the implementation of QL, which includes extensible components for semantic analysis, translation, and interpretation. Our experience has led us to extract a small set of generic tools for language implementation, called *Naked Object Algebras*, which can be found online here: `https://github.com/cwi-swat/naked-object-algebras`.

## 2. Interfacing With Parsers

Practical language implementation requires interfacing with parsers. A (generated) parser has to invoke the necessary factory method during parsing to instantiate a program into a concrete algebra. However, this would require reparsing the input program for every operation that is needed.

A solution is to merge operations into combined interfaces (e.g., IEval and IPrint). The parser would then build multiple operations at the same time. Unfortunately, this requires advanced type system features, not present in languages such as Java [3]. Here we present a more modest proposal: the object structure is rebuilt for each operation, but the parser is invoked only once.

The recorder combinator shown in Figure 1 turns an algebraic interface into a generic "recorder" algebra. Calling factory methods on this algebra will create a structure that can recreate the same structure into another algebra. Intuitively, the recorder combinator is used as follows:

```
Build<X> builder = parse(recorder(Alg.class), input);
IEval eval = builder.build(new Eval());
IPrint print = builder.build(new Print());
```

The parser is invoked with the generic recorder algebra, derived from the Alg interface. The result is a builder object which has "recorded" the syntactic structure of the input. The builder can then be used to instantiate this fixed structure into different operations using the concrete algebras (e.g., Eval and Print).

## 3. Combining Languages

Object Algebras are primarily focused on language extension in the narrow sense. An example is extending a language for expressions with another type of expression. Language combination, on the other hand, involves combining two different types of languages, such as, for instance, expressions and statements.

Although inheritance could be used for language combination, in practice this leads to inflexible designs. Inheritance prevents operations to be specific for a sub language (e.g., expressions or statements). Furthermore, single-inheritance is limiting for combining independent extensions.

```
static <T> T union(Class<T> ialg, Object ...algs) {
  return (T)Proxy.newProxyInstance(ialg.getClassLoader(),
      new Class<?>[] { ialg },
      (x, m, args) -> {
        for (Object alg: algs)
          try { return m.invoke(alg, args); }
          catch (Throwable e) { }
        throw new Exception("method not found");
      });
}
```

**Figure 2.** Generic union combinator

To obtain more modular designs, we would like to keep the interfaces for statements and expressions separate, so that operations can be scoped to each sub language, and so that each sub language can be extended separately. For instance, the interface for statements could then be:

```
interface StmtAlg<E, S> {
  S assign(String x, E e);
  S ifThen(E c, S b);
}
```

The dependency of statements on expressions is captured in generic type E, but not on any concrete interface.

It is possible to combine language through delegation [2]. A single class implements both languages' interfaces and then dynamically delegates factory method invocations to their respective algebra. To avoid writing delegation methods for every combination of languages, the delegation can be handled using Java's dynamic proxies. This is realized by the union combinator shown in Figure 2. The union operator can be used to dynamically combine any number of languages.

The first argument of the union function is the class of a combined interface, i.e. an interface extending all the language interfaces we are about to combine. The rest of the arguments to union are any number of algebras (i.e. implementations of factory interfaces), that *together* realize the combined interface of the first argument. The dynamic proxy then intercepts factory method calls, and delegates them to the first algebra that can handle it. (We assume there is only one unique object algebra that can service the request.)

Here is an example of using union to combine expressions and statements for the operations evaluation of expressions (Eval), and execution of statements (Exec):

```
interface All<E,S> extends ExpAlg<E>, StmtAlg<E,S> {}
All<IEval,IExec> evalExec
  = union(All.class, new Eval(), new Exec());
```

The union combinator provides a flexible way of dynamically combining language interfaces. Furthermore, in combination with the recorder combinator, interfacing with a parser is for free: just pass evalExec to the build method on the builder object obtained using recorder.

## 4. Extensible Concrete Syntax

The union combinator facilitates combining the abstract syntax of languages, but it does not solve the problem of combining the concrete syntax of these languages. In this section we present a lightweight solution to this problem based on decorating factory methods with annotations that specify the concrete syntax of each language construct. An example is shown in Figure 3.

```
interface ExpAlg<E> {
  @Syntax("exp = NUM");
  E lit(int n);

  @Syntax("exp = exp '+' exp") @Level(10)
  E add(E l, E r);
}

class Tokens {
  @Token("[0-9]+")
  static int num(String s) { return Integer.parseInt(s); }

  @Token("[ ]+") @Skip
  void ws();
}
```

**Figure 3.** Specifying concrete syntax using annotations

The `@Syntax` annotation contains a syntax production[2]. Each production needs to correspond to the signature of the factory method. That is, non-terminals should correspond to the generic arguments of the method. Non-generic arguments map to tokens. For instance, the **int** `n` argument of the `lit` constructor maps to the `NUM` token. Tokens are defined in a separate interface as static methods from `String` to the actual token value (e.g., of type **int**). Token methods representing whitespace or comments may be annotated with `@Skip`. Finally, to deal with operator precedence, factory methods can be annotated with `@Level` annotations indicating ordering constraints on the productions.

The parser generator reflectively collects all syntax annotations for a certain set of interfaces, and generates a parser for that specific combination of languages. The generated parser will call the appropriate factory and token methods to create the desired structure.

## 5. Case Study: QL

QL is a simple language for defining questionnaires. An example questionnaire is shown in Figure 4. QL programs are rendered as an interactive form. The rendering of the questionnaire in Figure 4 is shown in Figure 5. The language features labeled, typed questions, which can be answerable or computed. In the latter case, the value of a question is computed in terms of what the user of the questionnaire answered to other questions. Questions can be made conditional using an if-then-else construct.

The implementation of QL realizes the following features:

- *Syntax*: language interfaces, all annotated with concrete syntax productions for parsing.
- *Check*: name resolution, undefined name checking and type checking of expressions and statements.
- *Eval*: evaluation of expressions.
- *Format*: pretty printing, including insertion of parentheses where needed.
- *Render*: present the questionnaire as an interactive form; entered values are propagated to dependent, computed questions.

There is no inheritance. All combination of languages happens through `union`. Parsing-based instantiation into different operations is done using the `recorder` algebra. The full code of the case-study can be found here: `https://github.com/cwi-swat/ql-obj-alg`.

[2] Currently, we use ANTLR4 as parser generator, and the syntax of production closely follows the syntax used in ANTLR4.

```
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 4.** Example QL questionnaire

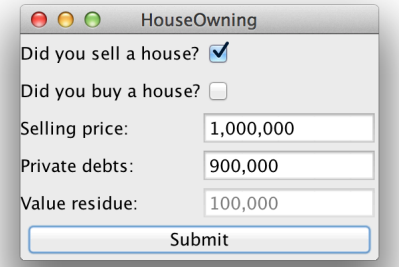


**Figure 5.** Rendered QL questionnaire

***Language Combination*** To illustrate the combination of `recorder` and `union`, the following method could be used for formatting questionnaires:

```
void prettyPrint(InputStream input, StringWriter output) {
  IAllAlg recorder = recorder(IAllAlg.class);
  Builder builder = (Builder) parse(input, recorder);
  FormFormat ff = new FormFormat();
  StmtFormat fs = new StmtFormat();
  ExprPrec prec = new ExprPrec();
  ExprFormat<ExprPrecedence> fe = new ExprFormat<>(prec);
  IAllAlg all = union(IAllAlg.class, ff, fs, fe);
  IFormat printableForm = builder.build(all);
  printableForm.format(0, false, output);
}
```

First a `recorder` is created for the `IAllAlg` interface which extends the expression, statement and form interfaces. The input is then parsed, which results in a builder object. Then the form, statement and expression specific implementations of formatting are instantiated. Note that formatting of expressions (`ExprFormat`) depends on an interpreter computing precedence levels (`prec`). Finally, the three algebras (`ff`, `fs`, and `fe`) are combined using `union`. The resulting algebra is input to the `builder` object to create the program structure that supports pretty printing (`printableForm`). The `format` method in interface `IFormat` performs the actual formatting starting with zero indentation and in horizontal mode (**false**).

***Language Extension*** To evaluate the extensibility of the base QL implementation, three additional, separate projects were created. The first project extends QL with an additional expression type (Modulo). The second project extends QL questions with an input validation construct (Validate). Finally, both extensions were combined.

The results in terms of source lines of code (SLOC) are shown in Table 1. The two independent extensions required very few lines of code, directly proportional to how much code is needed for a feature. The combination of both extensions (not shown) required

```
class Stmt2Box<X> implements StmtAlg<X,X> {
  BoxAlg<X> ba;
  X assign(String x, X e) {
    return ba.H(1, ba.L(x), ba.L(":="), e);
  }
  X ifThen(X e, X b) {
    return ba.V(0, ba.H(1, ba.L("if"), e), ba.I(2, 0, b));
  }
}
```

**Figure 6.** Translating statements to Box

**Table 1.** SLOC per feature QL and its extensions

| Feature | Base QL | +Modulo | +Validate |
|---|---|---|---|
| Syntax | 79 | 12 | 29 |
| Check | 993 | 24 | 148 |
| Eval | 1075 | 29 | 42 |
| Format | 335 | 32 | 40 |
| Render | 534 | — | 46 |
| *Total* | 3016 | 97 | 305 |

only a few lines of glue code to tie the base language and the two extensions together. This really showed the convenience of union.

The prettyPrint method above in the extension project for modulo includes the following additional statements to include pretty printing of modulo expressions:

```
IAllAlgWithMod recorder = recorder(IAllAlgWithMod.class);
...
ExprPrecWithMod precm = new ExprPrecWithMod();
ExprFormatWithMod fm = new ExprFormatWithMod(precm);
IAllAlgWithMod all = union(IAllAlgWithMod.class, ff, fs, fe, fm);
```

The only difference in constructing the pretty printer is to instantiate the recorder over the extended syntactic interface (IAllAlgWithMod) and adding the pretty printing algebra for modulo expressions to the union operator.

***Final Observations*** The code for formatting QL programs was implemented through translation to a generic Box algebra for pretty printing [1]. Each language construct is mapped to an L (literal), V (vertical) , H (horizontal) or I (indented) box. The resulting Box structure can then be pretty-printed to text. Directly delegating factory methods to a different algebra turns out to be a very powerful pattern to implement transformations, translations and desugarings. An example of this pattern using the Box algebra is shown in Figure 6. Note that the translation is generic: it does not depend on any specific operation interface.

We have performed some initial performance benchmarks. For all practical purposes, the overhead of dynamic proxies is negligible. Multiple builds after parsing necessarily incur a performance penalty, linear in the size of the constructed program. An experiment on an artifically constructed questionnaire of 1.4MB (source code) showed that parsing and directly creating a single operation takes roughly 0.125 seconds, whereas parsing and then building the same operation after parsing using the recorder algebra takes about 0.1875 seconds.

***Summary*** Summarizing, we consider the implementation of QL using the Naked Object Algebras framework to be very successful. The implementation of QL is independently extensible in both dimensions identified by the expression problem. The recorder and union combinators work together seamlessly, and provide a lot of flexibility in how algebras are combined and instantiated. Taking this approach to its limits has potential for constructing highly modular libraries of language building blocks, from which custom languages can be composed at will. Finally, even though the combinators presented here are very dynamic, the object algebra implementations themselves (Eval, Exec etc.) are statically typed.

## 6. Conclusion

The extensible and modular implementation of languages has the potential to turn software languages into product lines. Object Algebras support this vision by allowing language implementations to be extended with new operations or new language constructs. In this paper we have shared our experience in applying this technique in the implementation of a simple DSL, QL.

We have extracted our experience into the Naked Object Algebras framework, a generic toolbox for the flexible combination of algebras, interfacing with a parser and supporting extensible concrete syntax. The result can be considered as a first step towards a comprehensive toolbox for practical, end-to-end implementation of extensible languages.

## References


[1] J. Coutaz. The box, a layout abstraction for user interface toolkits. Technical Report 2127, Carnegie Mellon University, 1984.

[2] B. C. d. S. Oliveira and W. R. Cook. Extensibility for the masses: practical extensibility with Object Algebras. In *ECOOP'12*, pages 2–27. Springer, 2012.

[3] B. C. d. S. Oliveira, T. Van Der Storm, A. Loh, and W. R. Cook. Feature-oriented programming with object algebras. In *ECOOP'13*, pages 27–51. Springer, 2013.

[4] P. Wadler. The expression problem. Online, November 1998. `http://www.daimi.au.dk/~madst/tool/papers/expression.txt`.