Evolving Languages with Object Algebras

Pablo Inostroza
Centrum Wiskunde & Informatica (CWI)
Amsterdam, The Netherlands
Email: pvaldera@cwi.nl

Tijs van der Storm
Centrum Wiskunde & Informatica (CWI)
Amsterdam, The Netherlands
Email: storm@cwi.nl

***Abstract*—Object Algebras are a programming technique for the extensible implementation of recursive data types. This extended abstract introduces Object Algebras and shows how they could be used to develop highly evolvable software languages. The paper is concluded with a discussin of directions for further research.**

I. Introduction

Object Algebras [4] are a solution to the expression problem [7]. This means they support the extension of a data type along two dimensions: data type variants and the operations over the data type. Since the abstract syntax of a language is naturally described using recursive data types, this suggests that Object Algebras are a viable technique for implementing extensible language implementations.

Using Object Algebras, the abstract syntax of a language is described using generic factory interfaces. The following interface describes a simple language for expressions:

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

Operations over the abstract syntax are represented by implementations of such generic interfaces, where the type parameter (e.g., `E`) is bound to the type of the operation. For instance, evaluation of expressions can be realized as follows (using Java 8 closures):

```
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 type of the operation we're defining. The class `Eval` is a factory for interpreters of expressions.

To evaluate an expression it should be created using the `Eval` factory. As an example, the following generic method creates the expression "1 + 2" over any algebra `alg`:

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

To create evaluable expressions, one would call this method with an instance of `Eval`.

II. Adding Syntax

Any language operation is realized by (re)implementing the generic factory interface. To add another language construct, however, we need to extend the generic factory interface itself first. For instance, the following interface could represent the extension with multiplicative expressions:

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

Existing operations can then be extended by implementing the extended interface and subclassing the class representing the base operation. For instance, to extend the evaluator of expressions to support multiplication, one would write the following class:

```
class EvalMul extends Eval implements MulAlg<IEval> {
  IEval mul(IEval l, IEval r) {
    return () -> l.eval() * r.eval();
  }
}
```

Expressions should now be created over the extended interface `IMulAlg` using the factory `EvalMul`.

III. Changing Semantics

Sometimes we do not want to add a new language construct, but change the semantics of an existing construct, for instance, to fix a bug. This can be achieved using plain inheritance.

Consider the contrived example of changing the behavior of `add` to perform subtraction instead of addition. This can be expressed by overriding the constructor method `add` of `EvalMul`:

```
class SubIsTheNewAdd extends EvalMul {
  IEval add(IEval l, IEval r) {
    return () -> l.eval() - r.eval();
  }
}
```

IV. Advice

If it's not needed to completely replace the semantics of a construct, we can inherit from an operation and call **`super`** to selectively add "advice" to language constructs, as a simple form of Aspect-Oriented Programming (AOP) [3]. .

For instance, let's say we decide that the `add` construct is deprecated. In this case we want to keep the original behavior[1] but issue warning message to the user:

[1]"Extend and deprecate" is a common language evolution pattern.

```
class DeprecateAdd extends SubIsTheNewAdd {
  IEval add(IEval l, IEval r) {
    System.err.println("WARNING: + is deprecated");
    return super.add(l, r);
  }
}
```

Note that it is equally possible to additionally wrap `l` and `r`, and capture the result of calling **`super`**.

V. Desugaring

A common strategy to extend a language with a new construct is by "desugaring" it to a combination of existing constructs. Object Algebras in combination with Java 8 default methods provide a natural way of specifying such extensions.

Consider the addition of unary negative expressions `-x`. Semantically, this is equivalent to `-1 * x`:

```
interface NegAlg<E> extends MulAlg<E> {
  default E neg(E e) {
    return mul(lit(-1), e);
  }
}
```

The default method provides a default implementation of the `neg` constructor. This works for every operation implemented over `NegAlg` (i.e., for every binding of `E`). If the desugaring is undesired, for instance when pretty printing, `neg` can always be overridden in the concrete implementation of the operation.

VI. Concrete Syntax

The extension examples shown above all involved the abstract syntax of a language as modeled by generic factory interfaces. In a realistic language implementation, however, the concrete syntax should be accounted for as well. A pragmatic solution to this problem was presented in [2]. This solution is based on using Java annotations on factory methods to specify syntax productions. Using reflection all productions can be collected and used to generate a parser for a concrete parser generator (e.g., ANTLR4).

The syntax for the basic expression language is then specified as follows:

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

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

The `@Level` annotation specifies the expression's precedence level. This information can also be used to guide pretty printing.

VII. Discussion

This extended abstract introduced Object Algebras as a technique for evolving language implementations. However, Object Algebras are very young; there's not much experience yet on how to apply them in realistic case studies (but see [2]). In this section we discuss ongoing research and sketch out directions for future work.

a) Morphing Operations: Object Algebras support the extension of an operation to accomodate a new language constructs. However, it seems impossible to change the type of operations themselves. For instance, we cannot change the return type or parameter list of `eval` in `IEval`. Such changes would require reimplementing the evaluator for all cases. Additional parameters can be sometimes simulated using side effects in fields of in the concrete algebras. However, this might require maintaining a stack to simulate parameter passing. This is both tedious and error-prone. This problem is particular relevant when a language is extended with additional effects (e.g., state, continuations, backtracking, etc.).

b) Program Analysis: Object Algebras map syntactic constructs to denotations (objects) that represent the desired semantics. Often the result is simply an interpreter. Further research is needed how to specify complex program analyses (e.g., name resolution, type inference, data flow analysis, etc) as Object Algebras in an extensible way. Recent work on the relation between Object Algebras and attribute grammars also shows promise in this regard [6]. Another approach to investigate is the framework of abstract interpretation [1]. In this case a the syntactic constructs are not mapped to a concrete semantics, but to an abstract semantics (e.g., representing types).

c) Cross-cutting Concerns: Language implementation is full of cross-cutting concerns. Examples are: tracing, profiling, unique identities for name analysis, origin tracking for error messages, inserting hooks for debuggers. First steps towards generic advice in the context of object algebras is presented in [5]. Dynamic proxies are expected to be very valuable here, since they allow us to implement any operation interface (e.g., `IEval`) dynamically. However, their applicability is hampered if the signature of the operations needs to change (cf. previous point).

VIII. Conclusion

Object Algebras show a lot of promise for the implementation of evolvable languages. However, more research is needed to make their essential ingredients more modular and composable. Finally, real-life case studies are required to go beyond the stage of simple expression-oriented languages.

References

[1] P. Cousot. Abstract interpretation. *ACM Comput. Surv.*, 28(2):324–328, 1996.

[2] M. Gouseti, C. Peters, and T. van der Storm. Extensible language implementation with Object Algebras (short paper). In *GPCE'14*, pages 25–28. ACM, 2014.

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. *Aspect-oriented programming*. Springer, 1997.

[4] 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.

[5] 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.

[6] T. Rendel, J. I. Brachthäuser, and K. Ostermann. From Object Algebras to Attribute Grammars. In *OOPSLA'14*, pages 377–395. ACM, 2014.

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