Solving the Expression Problem for LTL using Object Algebra

René Kremer, Hannes Kallwies, and Simon Thiessen

Institute for Software Engineering and Programming Languages, Universität zu Lübeck, Lübeck, Germany
rene.kremer@student.uni-luebeck.de
hannes.kallwies@student.uni-luebeck.de
simon.thiessen@student.uni-luebeck.de

Abstract. The expression problem, as described in [1], describes the problem of defining data types, which can be extended with new data types and new functions without recompiling existing code and retaining static type safety. To handle this problem new paradigms and patterns were developed. In this paper we will discuss the use of object algebra as an alternative to patterns like the interpreter or visitor pattern. Linear Temporal Logic (LTL) [3] will be used as an example to show how one can use object algebra to solve the expression problem. Starting with a given set of LTL expressions and an evaluation function the LTL object algebra will be extended by a new data type and a new function.

Keywords: expression problem, visitor pattern, interpreter pattern, object algebra

1 Introduction

The expression problem was coined by Philip Wadler in [1] and describes a problem, which can be used to benchmark the expressiveness of programming language by a question such as "how much can your language express?". The expression problem therefore discusses strengths and weaknesses of programming languages and paradigms.

Philip Wadler presented in [1] the goal to define data types that could be extended by new cases of data types and new functions, while retaining static type safety and without recompiling existing code. An extra requirement was added by [2] and mentions the *independent extensibility*, which means that it should be possible to combine independently developed extensions.

There are different approaches to solve the expression problem, e.g. the visitor pattern and interpreter pattern, which will be introduced in section 2. As with nearly everything each approach has it's advantages and disadvantages.

The expression problem itself can be part of different kinds of practical problems one needs to solve, e.g. the string matching with regular expressions. In that particular case the interpreter pattern would suggest different classes for expression such as *literal*, *alternation* and *repetition* as shown in [4].

In practical terms regular expression will mostly be modeled via a state machine instead of the usually more complex expression classes. But one could easily come up with their own grammar which needs to be interpreted to get the information it expresses, e.g. the commands of RoboCups 2D Soccer server[1].

In section 2 we will give more insight about the visitor and interpreter patterns as well as object algebra. Our approach using object algebra with a given set of LTL expressions and their extended types and functions will be described in section 3. The conclusion in section 4 compares object algebra and its strengths and weaknesses to the visitor and interpreter pattern.

2 Approaches to the expression problem

In this section we will give a short overview of how the patterns work for a comparison later on. We will start with the interpreter pattern in subsection 2.1, move on to the visitor pattern in subsection 2.2 and finish this section with object algebra in subsection 2.3.

2.1 Interpreter Pattern

The interpreter pattern is used if a certain kind of problem occurs often enough. Instances of the problem will then be expressed in a simple language. The interpreter can then solve the problem by interpreting the sentences of the language.

In object-oriented languages this can be done by writing classes which represent expressions of the language. [4] uses the language for regular expressions as an example.

Therefore every regular expression is part of a grammar defining possible expressions, which can be built into an abstract syntax tree (AST). An example AST is shown in Figure 1.

Operations will be implemented in the expression class. That means that if we want to evaluate an expression, each node of the AST evaluates itself with the defined operation.

The following benefits and liabilities are described in [4]:

It's easy to change and extend the grammer: Using classes to represent grammar rules allows the usage of inheritance to extend the grammar. New data types will be derived from existing ones.

Implementing the grammer is easy, too: The nodes of the AST have similar functionalities, which makes the implementation easy and repetitive. This process can be automated with compiler or parser generators.

Complex grammars are hard to maintain: At least one class is defined for every expression type. This means that complex grammars have many different classes, which need to be maintained. Adding a new operation on these classes means one needs to change every class accordingly.

[1] http://www.robocup.org/

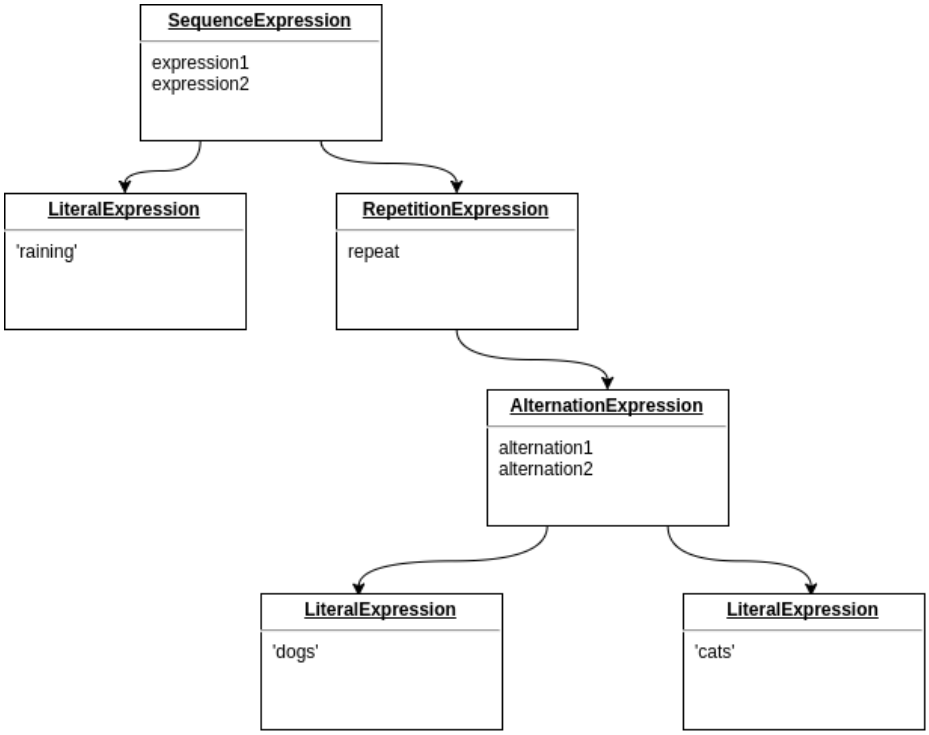

Fig. 1. Example in [4] for the AST of regular expressions using the interpreter pattern

So as one can see the interpreter is great for grammars that are rather simple and don't change much. Extending the grammar is easy, but adding new operations is not. Object algebra uses the terms *data variation* and *operations*. By using these terms one can briefly summarize the interpreter as a good way to handle data variations to a certain extend (not too complex grammars), but adding new operations to expressions can easily lead to many changes as every class of each expression needs an implementation of the new operation.

2.2 Visitor Pattern

Unlike the interpreter pattern the intent of the visitor pattern is to represent operations, which will be performed on the nodes of an AST [4].

With the visitor pattern, you define two class hierarchies. One is for the elements (or data variations) being operated on and one for the visitors that define operations on the elements [4].

That means that if one wants to add a new operation there needs to be a new visitor class.

Data variations need a function to *accept* a visitor. This accept function is no more than a callback, which calls the *visit* function of the visitor, where the

parameter is the data variations object. With that information the visitor knows which kind of node of the AST he is visiting and which exact function he needs to call to operate on this particular node. An example of this mechanic is shown in Figure 2.

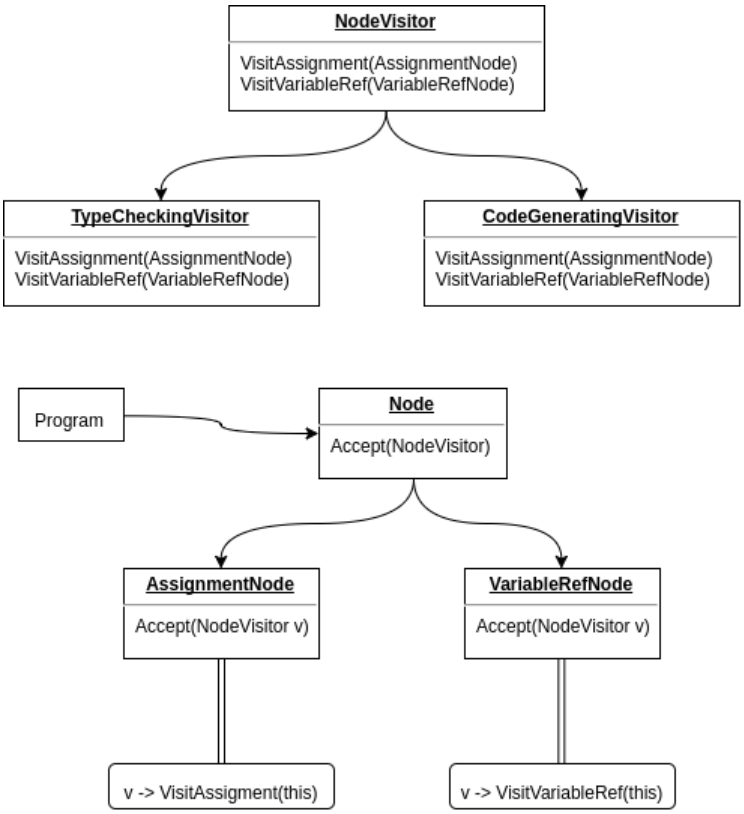

Fig. 2. Example in [4] for the visitor pattern

In [4] the following benefits and liabilities are described for the visitor pattern:

Visitor makes adding new operations easy: New operations are defined as new visitor classes. This way one doesn't need to change each class of your object structure.

A visitor gathers related operations and separates unrelated ones: As a visitor represents an operation over a node of the AST, the operations are separated in different visitor classes. That means that the basic *separation of concern* principle is handled by design.

Adding new ConcreteElement classes is hard: On the other hand, adding a new data variation is hard, as each visitor needs to handle this new data

variation. Additionally the data variation needs to *accept* each visitor as well. This is not suited for structures with frequent changes of data variations and their hierarchy.

Visiting across class hierarchies: Iterators can't work across object structures with different types of elements. The visitor however can because of the visit functions, which accept certain types of data variations. These data variations don't have to be related through inheritance.

Accumulating state: The state of a data variation is implicitly passed as an argument to the visitor. Therefore there is no need for extra arguments or global variables in the visit function.

Breaking encapsulation: The visitor pattern assumes that the interface of the data variations lets the visitors do their job. This might lead to public functions that allow access to the internal state of a data variation.

The *double-dispatch* mechanism (accept and visit function) mentioned in [4] is required if the visitor is called by the data variation. Terence Parr showed in [5] how one can create an independent visitor, that distinguishes the different nodes of an AST on the token type, which ANTLR uses in the grammar. Switching between the types allows the visitor to traverse the AST and identify each node.

This external visitor has, according to [5], some advantages as the method names can be named relevant to the task (instead of *visit*) and there is no need for an interface for the visitor. The double-dispatch mechanism is also encapsulated in the visitor and not in the data variation and operation (which is here represented by the visitor).

2.3 Object Algebra

Another solution to the *expression problem* was proposed by [6] and uses *object algebra*. Object algebra is based on *algebraic signatures* [7] and only uses *constructors*, which return values of the abstract set. The abstract set in object algebras is based on *data variants* and *operations*.

An example of these variants and operations is shown in Figure 3 for our LTL set. So every operation is available to every data variant. Implementing object algebras is done by creating a generic interface whose parameter is the abstract type [6]. The object algebra therefore declares generic factory interfaces for the variants and factories with the variant as type for implementing the operations. This will be shown more detailed in section 3 with the example set of LTL.

Object Algebra is similar to the *visitor* pattern as both decouple *variants* and *operations*. The visitor pattern has visitor classes for operations and the AST nodes as variants while object algebra uses generic factory interfaces to represent expressions (variants) and factories for operations. The main difference is that the visitor pattern focuses on operations. Object algebra uses the factory style to encapsulate the operations of an abstract expression. So to add new variants one needs to extend the interface of the generic factory, which is a natural and easy object-oriented approach. To add new operations one needs to implement

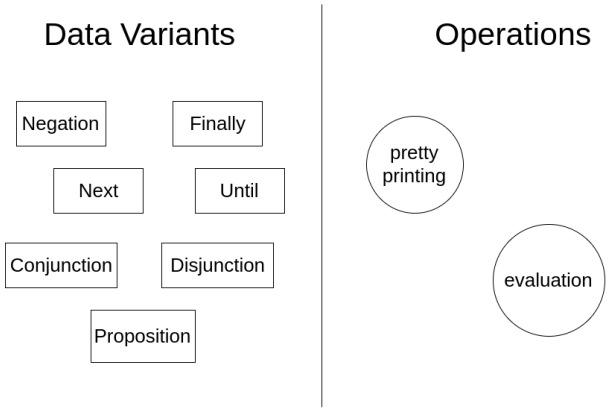

Fig. 3. Example of data variants and operations used in object algebra

this generic factory interface. So the techniques used are simple object-oriented ones. This also allows for the usage of *polymorphism*.

To summarize it:

Data type (data variant): The data variant is a generic factory interface or at least a method in the factory.
Operation: The operation is a factory implementation.
Adding new variants: Adding new variants is done by extending the interface.
Adding new operations: Adding new operations is done by implementing the interface.

3 Object Algebra to the LTL Expression Problem

In this section we will show how one can handle the LTL expression problem using object algebra. The data variants and operations are the same as shown in Figure 3. The code is public on github[2].

3.1 LTL Set

The set of LTL expressions includes *conjunction, disjunction, negation, next, until, finally, atomic propositions*. To show the extensibility of object algebras *finally* was added later in the development.

The operations on these expressions started with *pretty printing* and were extended with the operation to *evaluate* a LTL formula. The extensions of *evaluate* also shows the extensibility of operations in terms of object algebra.

[2] https://github.com/hadesrofl/oaltl

3.2 Implementation

The implementation uses an *ObjectAlgebraFactory*[3] to force the implementation of the data variants. So this abstract interface handles the data variants. The operations have an interface e.g. *PrintingObject*[4] for the pretty printing operation. This interface defines all needed functions with its parameters in an abstract way so that the *PrintingObjectFactory*[5] needs to implement these functions of the *PrintingObject* for each data variant of the *ObjectAlgebraFactory*.

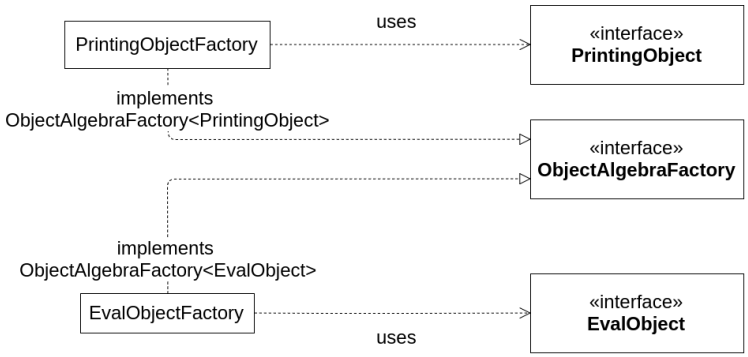

Fig. 4. Overview of the classes needed for the LTL implementation using object algebra

One can see that using this structure of object-orientation encapsulates the exact implementation of operation and data variants in a single factory for each operation. The interfaces ensure that all implementing classes are forced to implement the defined variants and also the operation.

In our example we ended up with adding the *finally* variant in the *ObjectAlgebraFactory*, but instead of this way to handle the extension, one could also just extend *ObjectAlgebraFactory* by something like *FinallyAlgebraFactory*. So the *PrintingObjectFactory* for example would implement *FinallyAlgebraFactory<PrintingObject>* and therefore all data variants defined in ObjectAlgebraFactory as well as the new *finally* addition.

4 Conclusion

To summarize subsection 2.1, the interpreter pattern is good for adding new data variants but adding new operations is difficult. Also the operations are scattered as they are implemented by each node of the AST. So the big drawback is the focus on implementing data variants and the operations being implemented in each node.

[3] Github: ObjectAlgebraFactory.java
[4] Github: PrintingObject.java
[5] Github: PrintingObjectFactory.java

As mentioned in subsection 2.2 the visitor pattern focuses on operations and therefore is good for adding new operations. It is also possible to read the internal state of the node the visitor is processing and generate an operation state out of this information. On the other hand it is hard to add new variants. Another point is that it generally uses the *double-dispatch* mechanism, which needs the variants to implement an *accept* method additionally to the already needed *visit* method. Also some functions or informations of the variants need to be public so that the visitor can do his work, which breaks the encapsulation of these element information.

In section 3 the general features and advantages of object algebra were shown. In terms of the expression problem it allows for a good (independent) extensibility while using the languages features to retain static type safety. The implementation in subsection 3.2 showed how one can use object algebra and that it is not that difficult to add new operations and variants. Using simple object-oriented techniques (e.g. polymorphism) is enough to use the potential of object algebra. Adding new variants and operations is easily done via generic interfaces for data variants and factories for the operations. Maintenance in a shallow object hierarchy is easy as the generic interface and the factories are the core of this approach. Maintaining a deep hierarchy of extensions (for operations or variants) might be a bit harder, though this is a general problem of software-development as more files lead to a bigger code base to maintain.

References

1. Wadler, P.: The Expression Problem. E-Mail Discussion (1998), `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`
2. Odersky, M., Zenger, M.: Independently Extensible Solutions to the Expression Problem. In FOOL'05
3. Pnueli, A.: The temporal logic of programs. In 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977, pages 46-57, IEEE Computer Society, 1977
4. Gamma, E., Helm R., Johnson R. and Vlissides J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series, Pearson Education, 1994
5. Parr, T.: Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages. Pragmatic Bookshelf, 2009
6. Oliveira, B., Cook, W.: Extensibility for the Masses - Practical Extensibility with Object Algebras. In ECOOP 2012 – Object-Oriented Programming: 26th European Conference, Beijing, China, June 11-16, 2012, pages 2-27, Springer Berlin Heidelberg
7. Guttag, J., Horning, J.: The algebraic specification of abstract data types. In Acta Informatica Vol. 10, pages 27-52, March 1978