The expression problem, Scandinavian style

Erik Ernst

Dept. of Computer Science, University of Aarhus, Denmark
eernst@daimi.au.dk

ABSTRACT

This paper explains how higher-order hierarchies can be used to handle the expression problem. The expression problem is concerned with extending both the set of data structures and the set of operations of a given abstract data type. A typical object-oriented design supports extending the set of data structures, and a typical functional design supports extending the set of operations, but it is hard to support both in a smooth manner. Higher-order hierarchies is a feature of the highly unified, mixin-based, extension-oriented kind of inheritance which is available in the language gbeta, which is itself a language that was created by generalizing the language BETA.

Keywords: Expression problem, class composition, gbeta

1. INTRODUCTION

The expression problem has been defined as follows by Torgersen[1]: "Can your application be structured in such a way that both the data model and the set of virtual operations over it can be extended without the need to modify existing code, without the need for code repetition and without runtime type errors."

This paper presents a new approach to the expression problem, based on the support for higher-order hierarchies which is a feature of the inheritance mechanism in the language gbeta. It is very straightforward for programmers to express the desired extensions in this style, and moreover different extensions are highly composable.

The expression problem is about two-dimensional extensions: if an abstract datatype is modeled in an object-oriented style by means of classes whose methods are the operations on the datatype, then it is easy to extend the set of variants by writing another class. If the abstract datatype is modeled in the style of an SML `datatype` with a set of pattern matching functions as the operations, then it is easy to extend the set of operations by writing yet another pattern matching function. In both cases it is much harder to perform the *other* extension, i.e., to add a new operation in the object-oriented style or to add a new data structure in the functional style, in both cases because it is necessary to make changes in many existing entities rather than just writing one new entity. Space constraints imply that the traditional forms of the expression problem cannot be presented by detailed code examples here, but this has been done several times before.[1,2] Instead, we will proceed to describe and discuss the new solution.

Section 2 presents the running example in terms of which our solution to the expression problem is exposed, and gives the code for the initial design. This design is extended in Sect. 3 with a new operation, and in Sect. 4 with a new kind of data. Next, Sect. 5 shows how the two extensions may be composed and glued together. Finally, Sect. 6 briefly discusses related work and Sect. 7 concludes.

2. THE EXAMPLE PROBLEM

Following the tradition in relation to the expression problem, we will focus on a compiler-oriented problem, namely that of representing abstract syntax trees for a tiny language. The desired extensions will then correspond to adding a new operation on the abstract syntax trees, and adding a new kind of nodes in the trees. Here is the source code for the initial design:

```
class Lang {
  virtual class Exp {
    String toString() {}
  }
  virtual class Lit extends Exp {
    int value;
    Lit(int value) { this.value=value; }
    String toString() { return value; }
  }
  virtual class Add extends Exp {
    Exp left,right;
    Add(Exp left, Exp right) {
      this.left=left; this.right=right;
    }
    String toString() {
      return left.toString()+"+"+right.toString();
    }
}
```

Ex. 1

We use a syntactic style which is close to the Java programming language,[3] but which is in fact just a modified surface syntax for the language gbeta. In particular, class attributes may be virtual, which means that they may be redefined (more precisely: further constrained) in subclasses of the enclosing class, and this is what we will exploit in order to create the desired extensions later. Note that gbeta supports inheritance between virtual classes; in BETA such an inheritance relation is not supported, but gbeta supports it as a result of a deep generalization of the underlying concepts and mechanisms. This is the basis for higher-order hierarchies, which is described in detail elsewhere.[4]

The class `Lang` contains three classes `Exp`, `Lit`, and `Add`, the latter two being subclasses of the first one. The class `Exp` represents abstract syntax trees for expressions in the tiny language we are dealing with, and the two other classes are the only possible forms of expressions, namely integer literals, `Lit`, and addition expressions, `Add`. The only operation available in this basic version is `toString`.

3. ADDING A NEW OPERATION

We can extend the class family with a new operation in the following way (note that we need not edit the base family):

```
class LangEval extends Lang {
   refine class Exp {
     int eval() {}
   }
   refine class Lit {
     int eval { return value; }
   }
   refine class Add {
     int eval { return left.eval()+right.eval(); }
   }
}
```

Ex. 2

The effect of this is that we create a derived class family (a new version of `Exp`, `Lit`, and `Add`), each of them created by extending the version in `Lang` with the new `eval` method. The keyword `refine` is used to specialize an inherited virtual class attribute, and the semantics is that the virtual class attribute is constrained to be a subclass of the new declaration. In other words, no matter what class `Exp` denotes in `Lang`, it will denote that same class extended with the `eval` method in `LangEval`. Note that `Add` was declared to be a subclass of `Exp` in `Lang`; such a subclass relation is maintained even when virtual classes are refined, and this means that `Add` in

`LangEval` is a subclass of the *current* value of `Exp`, which is then extended with a particular implementation of the `eval` method.

Note that all we had to do in order to extend the entire family with a new operation was to declare which classes should be extended with the new method (`refine..`), and then declare the method. If `Lit` and `Add` could have used an implementation of eval written in `Exp` then we could have written just that single new method in a refinement of `Exp`, and all subclasses (in this case: `Lit` and `Add`) would have inherited the new method without any need to mention them explicitly. This is again because the declared inheritance relations are automatically maintained.

4. ADDING A NEW DATA STRUCTURE

Extending the class family with a new member is also easy:

```
class LangNeg extends Lang {
  virtual class Neg extends Exp {
    Neg(Exp exp) { this.exp=exp; }
    String toString() { return "-(" + exp.toString() + ")"; }
    Exp exp;
  }
}
```

Ex. 3

Here we create a new class family whose members have the same structure as in `Lang` (because there are no `refine` declarations), but a new family member is added, namely `Neg` which represents the unary negation operator. As before, there is no need to edit `Lang` in order to create this extension of it, and the new extended version of `Lang` is created simply by describing the delta—in this case the class to add to the family. Note, however, that the new family member is declared to be a subclass of `Exp`. This means that extensions to `Exp` will also added to `Neg`, as we shall see in the next section.

5. COMPOSING BOTH EXTENSIONS

It is possible to use the two extensions together, by composing the two class families created in Sect. 3 and 4. In gbeta, class composition is supported by means of the '`&`' operator, but since this character is already used for other purposes in Java syntax we will use the (non-ASCII) symbol $\oplus$ to play this role. The class families may then be combined in the following manner:

```
class LangNegEval extends LangEval ⊕ LangNeg {
  refine class Neg {
    int eval() { return -exp.eval() }
  }
}
```

Ex. 4

It is trivial to compose the two class families, producing a new family which contains the base material from `Lang` as well as the added `Neg` class from `LangNeg` and the added `eval` method from `LangEval`. To do this, we can just use `LangEval`$\oplus$`LangNeg`. However, we need to add a little bit of glue code to this combination, because `LangEval` does not know about the class `Neg` and `LangNeg` does not know about the method `eval`, and the result is that the implementation of `eval` for the class `Neg` is non-existent. This might be fine since `Neg` does in fact inherit `eval` from `Exp` in context of the combined family, but the implementation in `eval` in `Exp` is not suitable for `Neg`, so we have to add an implementation of `eval` specifically for `Neg`. This extension is achieved by the body of class `LangNegEval` above.

Note that it is not a problem with the expressive power of the language that forces us to write this glue code, it is a problem which is inherent in the combination of independent extensions. We could never expect the independently added method and the independently added class to match up in such a way that the combination of the extensions would know how to implement the new method for the new class—this is inherently an application domain dependent problem, which must be solved by a programmer who writes the missing method.

6. RELATED WORK

Krishnamurthi et al.[2] describe an extension of the visitor pattern with factory methods is used to ensure new datatypes can be added while maintaining consistency. Adding new operations is easy when using the visitor pattern, so this establishes a two-dimensional extensibility. However, it leads to significantly more complex programs than what we have shown in this paper, it does not support smooth composition of independent extensions, it is not well-integrated in the type system (it uses explicit type casts), and it does not support type-safe polymorphic usage of complete families of classes, also known as family polymorphism.

Torgersen[1] describes a number of visitor based approaches, with a similar level of complexity as in the previous approach and also without family polymorphism or extension composition, but it removes the need for dynamic casts. Moreover, in this case there is a feature which is not available in the approach we have shown: it is, under certain circumstances, possible to mix objects belonging to different families. E.g., an expression may contain nodes from the basic family (without negation), and this expression could then be used as the subtree of a `Neg` node.

Zenger and Odersky[5] describe how SCALA is used to express two different families of solutions to the expression problem which are capable of combining independently added extensions. Two extensions adding datastructures are combined, and two extensions adding operations are combined—and it is not clear whether two extensions can be combined (and glued) if one of them adds a datastructure and the other one adds an operation. SCALA uses some constructs similar to virtual types (it supports abstract type members in objects), and it is more theoretically well-analyzed but a little less expressive than `gbeta`. In particular, the example programs are more complex than the ones we have shown here, because SCALA does not support propagating combination and consequently the combination of nested type members must be spelled out manually.

7. CONCLUSION

We have presented an approach to the expression problem based on higher-order hierarchies. Using this approach, the expression problem becomes a small matter of writing the classes and/or methods which need to be added to a given class family, and it is even possible to combine independent extensions and add the missing glue. We believe that this is the smoothest known type-safe approach to the expression problem.

REFERENCES

1. M. Torgersen, "The expression problem revisited – four new solutions using generics," in *Proceedings ECOOP'04*, M. Odersky, ed., *LNCS ?*, pp. ?–?, Springer-Verlag, (Oslo, Norway), 2004. To appear.
2. S. Krishnamurthi, M. Felleisen, and D. P. Friedman, "Synthesizing ojbect-oriented and functional design to promote re-use," in *Proceedings ECOOP'98*, E. Jul, ed., *LNCS* **1445**, pp. 91–113, Springer-Verlag, (Brussels, Belgium), July 1998.
3. B. Joy, G. Steele, J. Gosling, and G. Bracha, *Java(TM) Language Specification (2nd Edition)*, Addison-Wesley Publishing Company, 2000.
4. E. Ernst, "Higher-order hierarchies," in *Proceedings ECOOP 2003*, L. Cardelli, ed., *LNCS 2743*, pp. 303–329, Springer-Verlag, (Heidelberg, Germany), July 2003.
5. M. Zenger and M. Odersky, "Independently extensible solutions to the expression problem," Tech. Rep. IC/2004/33, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, 2004.