The Expression Problem, Gracefully

Andrew P. Black
Portland State University
black@cs.pdx.edu

Abstract

The "Expression Problem" was brought to prominence by Wadler in 1998. It is widely regarded as illustrating that the two mainstream approaches to data abstraction — procedural abstraction and type abstraction — are complementary, with the strengths of one being the weaknesses of the other. Despite an extensive literature, the origin of the problem remains ill-understood. I show that the core problem is in fact the use of *global constants*, and demonstrate that an important aspect of the problem goes away when Java is replaced by a language like Grace, which eliminates them.

Keywords data abstraction, algebraic data types, rows and columns, procedural abstraction, objects, expression problem, extensibility.

1. Introduction

The Expression problem was given that name by Wadler in 1998 [9], and demonstrates the importance of a catchy name in securing the immortality of publication. Wadler did not claim to have invented the problem; on the contrary, he pointed out that it had already been widely discussed, notably by Krishnamurthi, Felleisen, and Friedman in an ECOOP paper published earlier that year [5]. Oliveira and Cook's later ECOOP paper [6] opens by declaring that

> The "expression problem" (EP) [4, 7, 9] 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.

It does not take much study of the voluminous literature on the expression problem to realize why it has proven so difficult to solve: the problem is over-constrained, in the sense that the problem statement itself rules out any possibility of a perfect solution. The various published solutions fill a three-dimensional space. Two dimensions are frequently discussed and compared. The first is the set of features offered by the implementation language (classes, class extensions, generic types, algebraic data, etc.); the second is the pattern employed by the programmer (abstract factory, visitor, object algebra, etc.). The third dimension, in contrast, is hardly mentioned: which of the constraints the authors choose to loosen to obtain a solution. To understand what these constraints are, we need to examine the problem in more detail.

2. The Problem

As described by Krishnamurthi, Felleisen, and Friedman [5], the expression problem considers a representation of some composite heterogeneous recursive structure, such as the expression tree of a programming language. There are operations (also called "tools") on those structural elements, for example, code generation, or pretty-printing. We want to be able to extend this structure in two orthogonal dimensions:

– the data dimension, in which we might add new variants to the composite data, for example, representations of new programming language statements, and

– the operation dimension, in which we might add new operations, such as interpretation.

Krishnamurthi, Felleisen, and Friedman's original constraint was that "ideally, these extensions should not require any changes to existing code". Wadler added the constraint of static type safety (e.g., no casts), and strengthened no *changes* to existing code to no *recompilation* of existing code. Torgersen, in his comprehensive survey of the problem and its solution using generics [8], observes that "existing code" includes not only the implementation of the expression tree, but also the *creation code* — the code responsible for creating instances of the structure — and the *client code*, which sends messages to those instances.

3. The Solutions

In a paper of this length, we cannot even *survey* the extensive literature on solving the expression problem. Instead, we will examine four interesting points in the solution space.

3.1 Smalltalk

Smalltalk systems solve the expression problem very neatly, by allowing modules to both add new classes, and to add features to existing classes. Ruby has the same property. The Smalltalk *language* itself does not have a concept of module, but every Smalltalk *system* does, and the ANSI Standard [1] specifies an "Interchange Format" intended for communicating modules between Smalltalk implementations.

Modules can contain class definitions, but can also contain *method* definitions and class *re*-definitions, which can be used to add methods and instance variables to existing classes. Thus, it is simple to extend a representation in the data dimension by writing a module that defines a new class, and equally simple to extend a representation in the operation dimension by writing a module that adds a method to each of the classes representing the variants of the composite structure.

This solution meets two out of the three constraints. First, no existing code need be changed — with the obvious exception that there must be some changes in the creation code to allow the new structural variants to be created, and in the client code so that the new tools can be employed. Second, independent compilation is preserved: Smalltalk compiles each method separately, so adding individual methods poses no difficulty. Of course, it fails to meet Wadler's constraint of static typing: Smalltalk does not have a static type system.

It is instructive to ask *why* this solution works: what property of Smalltalk makes it possible? It does not depend on Smalltalk running in an image or on meta-programming, although these techniques are certainly used in the implementation. The critical properties are that modules and classes are orthogonal, and that classes are named by global *variables*, not global *constants*. Thus, the loading of a new module can change the meaning of a class name.

Notice that there is nothing fundamental that stops this process from working with static typing — if we assume that all changes are extensions that create subtypes. (Clearly, *deleting* methods from existing objects would not be type-safe, but the expression problem is concerned with extension, not deletion.)

3.2 The Java Family of Languages

In Java as originally defined, the expression problem was unsolvable. Extension in the data dimension was easily achieved by a *package* that added a new class implementing a new kind of tree node, but extension in the operation dimension was more problematic. One might attempt to create new subclasses of all of the existing classes, each extending its superclass with the new operation. This fails for two reasons. The reason most discussed in the literature — and indeed the motivation for Wadler's original note and his introduction of the pun "expression problem" — is the limited expressivity of Java's type system: a limitation that GJ set out to remedy.

To understand the problem, imagine that the original tree contained (among others) a class Sum, representing a sum expression, with instance variables left and right, and that the only operation initially implemented by Sum and its siblings is pretty. The types assigned to these instance variables would naturally include only the pretty operation. To add an eval operation, we subclass Sum to create EvaluableSum, which adds an eval method, the body of which contains

```
{ return left.eval() + right.eval() }
```

Unfortunately, this won't type-check, because the static types of left and right are too restrictive. If we assume that the creation code uniformly instantiates the evaluable subtypes, then left and right will indeed contain evaluable objects, but Java's type system loses this information.

The basic idea behind the fix is to make the type of the instance variables a parameter. Then, without changing the source code, their declared type can be changed. Torgersen [8] works through the details using Java Generics. Unfortunately, the intricacies of Java's type system — F-bounding the parametric types, and then creating fixpoint subclasses to instantiate them — forces him to observe "that the initial simplicity of the . . . approach has disappeared".

In spite of all of this, the Java "solutions" fail for a second, completely different reason: they require us to change the original code. Although the machinery of type parameters may spare us from having to change the code in the base classes, no type machinery can spare us from changing the creation code. Why is this? In a normal Java program the creation code will create instances of the various classes of expression by applying the **new** operator to the global name of the class. The structure of the Java language says that we can't rebind that name to something else, and the restrictions of the expression problem say that we can't change the code in the expression classes.

It is the thesis of this paper that this is a major problem. The creation code is not something insignificant: in a compiler, for example, the "creation code" is the whole of the parser. A "solution" to the expression problem that requires that we edit every AST-node creation statement in the parser just because we have added a new optimization pass is no solution. In practice, this problem can be mitigated by the clever use of Java's **import** statement, which can be used to introduce short names to stand in for the global names of the base classes, but redefining these short names still requires recompilation of the creation code, thus violating one of Wadler's requirements for a solution.

3.3 Grace

Grace is a new object-based language that a small group of academics have been developing since 2010 [2]. Its target audience is novice programmers who are learning the essentials of object-oriented programming. The following features of Grace are relevant to the expression problem:

module "exp_base"

```
dialect "staticTypes"

type Value = Object
type Exp = { eval −> Value }

factory method lit(i:Number) −> Exp {
    method x −> Number { i }
    method eval −> Value { x }
}
factory method sum(a:Exp, b:Exp) −> Exp {
    method l −> Exp { a }
    method r −> Exp { b }
    method eval −> Value { l.eval + r.eval }
}
// Demonstration:
def threePlusFour:Exp = sum(lit 3, lit 4)
print "{threePlusFour} = {threePlusFour.eval}"
// prints:   an object = 7
```

Figure 1: The *exp_base* module in Grace: the base code that we will extend.

1. there are no global variables;
2. modules, implemented as files, become objects at runtime;
3. within a module, Grace is block-structured; and
4. modules are imported under a name chosen by the *client*.

If you would like to run the examples that follow yourself, you can do so in your web browser (Chrome or Firefox) at http://www.cs.pdx.edu/~grace/minigrace/exp. The code is at http://www.cs.pdx.edu/~grace/minigrace/expProb/.

Figure 1 shows a minimal version of a module defining basic expressions in Grace; the code is based on that of Oliveira and Cook [6], but simplified into idiomatic Grace. (The original code is in the appendix, together with an explanation of the changes.) The ruled box indicates a file, which is compiled into a module object containing the features defined therein. Grace treats the file as if it were bracketed by **object** { ... }; the expression **object** { ... } is an object constructor, which manufactures a new object each time it is executed.

In this case, the module object contains two types, from the declarations on lines 3 and 4, and two methods, from the declarations on lines 6–14. (The **def**inition on line 16 is not visible outside the module.) A **factory method** is a method that creates and returns a new object that contains the features in the factory method's body. In other words, **factory method** m { ... } is equivalent to

```
method m {
    object { ... }
}
```

So, for example, the factory method sum (which corresponds to Oliveira and Cook's add) creates a new object with methods l, r and eval.

module "exp_and_pretty"

```
dialect "staticTypes"
import "exp_base" as baseExp
type Exp = baseExp.Exp & type { pretty −> String }

factory method lit(i:Number) −> Exp {
    inherits baseExp.lit(i)
    method pretty { x.asString }
}
factory method sum(a:Exp, b:Exp) −> Exp {
    inherits baseExp.sum(a, b)
    method pretty { "{l.pretty} + {r.pretty}" }
}
// Demonstration:
def threePlusFour:Exp = sum(lit 3, lit 4)
print "{threePlusFour.pretty} = {threePlusFour.eval}"
// prints:   3 + 4 = 7
```

Figure 2: The *exp+pretty* module, which extends *exp* with a method pretty.

Line 16 and 7 demonstrate the use of these expressions. Notice that attempting to print threePlusFour outputs *an object*. This is the result of the default asString method of sum objects.

One other feature of Grace is important for our discussion: the code implementing the module does *not* give the module a name. This is left to the clients of the module, who can choose any name they like.

Figure 2 shows an extension to *exp_base* in the operations dimension — the dimension that is "difficult" for object-oriented languages. The **import** statement on line 2 gives the name baseExp to the module object from Figure 1. The module *exp_and_pretty* adds a pretty method (corresponding to Oliveira and Cook's print) to both of the variants of the composite. Notice that the type Exp in this module is a subtype of Exp in the *exp_base* module. Because of this, the requests of pretty on line 11 are not well-typed, because the methods l and r return baseExp.Exp, which has no method pretty.

I believe that this problem can be solved using **SelfType**, in a manner similar to that employed in Bruce's LOOJ [3]. The basic idea would be to change the type annotations on methods l and r in Figure 1 from Exp to **SelfType**, where **SelfType** denotes the declared type of the current object, here Exp. When baseExp.sum is inherited (on line 10 of Figure 2), the return types of the inherited methods l and r would still be **SelfType**, and this would again be interpreted to mean the declared type of the current object, now the enhanced type Exp declared on line 3 of Figure 2. However, the design of Grace's type-system is not yet mature enough for me to assert that this idea actually works out once all of the details are taken into account.

The extension in the data dimension is shown in Figure 3. This is straightforward, as we would expect for the

module "exp_and_pretty_and_bool"

```
dialect "staticTypes"
import "exp_and_pretty" as baseExp
type Exp = baseExp.Exp
type Value = Object

method sum(l:Exp, r:Exp) -> Exp { baseExp.sum(l, r) }
method lit(x:Number) -> Exp { baseExp.lit(x) }

factory method bool(b:Boolean) -> Exp {
    method x -> Boolean { b }
    method eval -> Value { x }
    method pretty -> String { b.asString }
}
factory method iff(c:Exp, t:Exp, f:Exp) -> Exp {
    method eval -> Value {
        if (c.eval) then { t.eval } else { f.eval }
    }
    method pretty -> String {
        "if ({c.pretty}) then {t.pretty} else {f.pretty}"
    }
}

def e3plus4:Exp = sum(lit 3, lit 4)
def e2plus6:Exp = sum(lit 2, lit 6)
def ett:Exp = bool(true)
def ifExpr:Exp = iff(ett, e3plus4, e2plus6)
print "{ifExpr.pretty} = {ifExpr.eval}"
// prints:    if (true) then 3 + 4 else 2 + 6 = 7
```

Figure 3: The *exp+pretty+bool* module, which extends the module *exp+pretty* with a new data variant for booleans.

"easy" dimension. Lines 3–7 give local (unqualified) names to features imported from baseExp. The factory method bool (line 9) defines the data variant for boolean literals, and the factory method iff (line 14) defines the data variant for if-then-else expressions. Here there is no typing problem, because the types baseExp and Exp are identical, and Grace uses structural, rather than nominal, types.

What of the creation code — can this be reused? The absence of global constants from Grace makes it fairly easy to do so. Modules that create instances of the composite structure must contain a statement like

```
import "exp_base" as exp
```

If this is changed to

```
import "exp_and_pretty_and_bool" as exp
```

then the balance of the creation code can be re-purposed without change. If we wish to strictly observe the requirement not to change the source code, then an alternative (which we will not follow here) is to rename the source files. In either case, the module must be recompiled.

3.4 Object Algebras

Oliveira and Cook addressed the issues of complex parametric types and the reusability of creation code using Object

module "objectAlgebra"

```
dialect "staticTypes"
import "exp_base" as exp
type Exp = exp.Exp

// define the Object Algebra machinery
type IntAlg<A> = {
    lit(x:Number) -> A
    sum(e1:A, e2:A) -> A
}
factory method intFactory -> IntAlg<Exp> {
    method lit(x:Number) -> Exp { exp.lit(x) }
    method sum(a:Exp, b:Exp) -> Exp { exp.sum(a, b) }
}
method mk3Plus4<A>(v:IntAlg<A>) -> A {
    v.sum(v.lit(3), v.lit(4))
}
// compare the above with the normal expression:
// def e3Plus4:Exp = sum(lit 3, lit 4)

// add pretty−printing to expressions "retroactively"
type Pretty = { pretty -> String }
factory method prettyFactory -> IntAlg<Pretty> {
    factory method lit(x:Number) {
        method pretty -> String { x.asString }
    }
    factory method sum(a:Pretty, b:Pretty) {
        method pretty -> String { "{a.pretty} + {b.pretty}" }
    }
}

// demonstration
def x = mk3Plus4(intFactory)
// print "{x.pretty} = {x.eval}"
//   fails: no method 'pretty' in object x
def s = mk3Plus4(prettyFactory)
// print "{s.pretty} = {s.eval}"
//   fails: no method 'eval' in object s
print "{s.pretty} = {x.eval}"
//   prints:    3 + 4 = 7
```

Figure 4: The *objectAlgebra* module, which is a translation of Oliveira and Cook's solution to the expression problem.

Algebras [6]. Their basic idea is to abstract over the creation of the composite structure: rather than actually building the tree for an expression, they instead define a method that will build the tree on demand. This "tree maker" method is parameterized by the factory method that will build the tree nodes with necessary operations. Object Algebras rely on parametric types, but not on self types or F-bounds.

Figure 4 shows Oliveira and Cook's code translated into Grace. The module *objectAlgebra* shows extension in the operation dimension. Lines 11–17 abstract over the base code imported on line 2. The factory method intFactory encapsulates all of the data variants: it constructs an "algebra" with a method for each of the data variants in the base. Method

mk3Plus4 on line 16 is a "lifting" of the normal expression construction code (given in the comment on line 19); to actually build an expression tree, it is necessary to apply mk3Plus4 to a suitable object algebra. On line 33, it is applied to intFactory; the resulting structure x has lit nodes and sum nodes that understand just the eval method. Consequently, they don't understand pretty, as suggested by the comments on lines 34 and 35.

The extension to pretty-printing is shown in lines 23–30, which define a second object algebra, prettyFactory; on line 36 it is used to build an expression tree. As suggested by the comments on lines 37 and 38, this tree comprises nodes that understand just the pretty method. Consequently, attempts to eval it will fail. However, once we have built both the evaluable tree x and the pretty-printable tree s, we can achieve our goal by using each for its intended purpose, as shown on line 39.

Of course, it is possible to create an algebra with more than one operation, and in practice programmers will probably do so. But the *extensibility* of object algebras comes by adding new operations in *new* algebras; an individual algebra is no easier to extend than the class that it subsumes.

Extension in the data dimension is straightforward but verbose. In addition to defining the new variant structures bool and iff, it is also necessary to define a new algebra type that extends IntAlg with factory methods for the new variants, and two more object algebra factories: intBoolFactory as an extension of intFactory, and intBoolPrettyFactory as an extension of prettyFactory. The details can be found in the original article [6].

In a theoretical sense, using object algebras to defer construction of the expression tree does allow the creation code to be reused. But in practice, programmers will not write their creation code using object algebras unless they are expecting to have to extend their code. Thus, I do not believe that object algebras provide for unanticipated extension. They achieve reuse only at the cost of requiring preplanning. Moreover, the cost is large: because a different tree is built for each operation, operations that update the tree must be simulated by simulating a store.

4. Independent Extensibility

An additional requirement is sometimes imposed on the expression problem: independent extensibility [10]. One of the referees of this article wrote:

> In real life, a much more common scenario than Fig. 1 followed by Fig. 2 followed by Fig. 3 would be like this. Some party A defines *exp_and_pretty*. Another party B independently defines *exp_and_bool*. A third party C finds those and wants to combine them to *exp_and_pretty_and_bool*. This should be possible so that C need only define pretty for bool (in addition to importing the two previous modules). Can Grace handle that?

The answer is yes; the solution can be found in the independentExtensability subdirectory at the previously-referenced URL. However, the Grace solution is not fully general: it works only when parties A and B happen to make their extensions in orthogonal dimensions, A adding a new operation and B adding new data. This is because Grace uses inheritance to add extensions in the operation dimension, and composition to add them in the data dimension. If both A and B added new operations, then combining them would require some form of multiple inheritance, which Grace presently lacks.

5. Conclusion

Wadler's interest in the expression problem was based on his search for expressive type systems. But, typing aside, the problem gives us insights into the dangers of global constants. It is not just Java's type system that is problematic; even more fundamental is Java's use of a global namespace for classes, and the fact that classes are immutable. This means that it is impossible to reuse creation code written in the "normal" way, in which objects are created by **new**ing a class. In contrast, Grace's lack of a global namespace means that all creation code must be written relative to a local name, which can later be re-defined. Thus, Grace permits extension without pre-planning.

Acknowledgments

I thank Jeremy Gibbons, Kim Bruce and the anonymous referees for their comments on this material, which have helped to improve the presentation substantially.

References

[1] ANSI. *Draft American National Standard for Information Systems — Programming Languages — Smalltalk*. ANSI, Dec. 1997. Revision 1.9.

[2] A. P. Black, K. B. Bruce, M. Homer, and J. Noble. Grace: the absence of (inessential) difficulty. In *Onward! '12: Proceedings 12th SIGPLAN Symp. on New Ideas in Programming and Reflections on Software*, pages 85–98, New York, NY, 2012. ACM. URL http://doi.acm.org/10.1145/2384592.2384601.

[3] K. B. Bruce. Some challenging typing issues in object-oriented languages. *Electr. Notes Theor. Comput. Sci.*, 82(7):1–29, 2003. . URL http://dx.doi.org/10.1016/S1571-0661(04)80799-0.

[4] W. Cook. Object-oriented programming versus abstract data types. In J. W. de Bakker, W. d. Roever, and G. Rozenberg, editors, *Proceedings REX Workshop/School on the Foundations of Object-Oriented Languages*, volume 489 of *LNCS*, pages 151–178. Springer Verlag, 1991. URL http://www.cs.utexas.edu/users/wcook/papers/OOPvsADT/CookOOPvsADT90.pdf.

[5] S. Krishnamurthi, M. Felleisen, and D. P. Friedman. Synthesizing object-oriented and functional design to promote reuse. In E. Jul, editor, *ECOOP'98 — Object-Oriented Pro-*

gramming, volume 1445 of *Lecture Notes in Computer Science*, pages 91–113. Springer, 1998. ISBN 978-3-540-64737-9. . URL http://dx.doi.org/10.1007/BFb0054088.

[6] B. C. d. S. Oliveira and W. R. Cook. Extensibility for the masses. In J. Noble, editor, *ECOOP 2012 — Object-Oriented Programming*, volume 7313 of *Lecture Notes in Computer Science*, pages 2–27. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-31056-0. . URL http://dx.doi.org/10.1007/978-3-642-31057-7_2.

[7] J. C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In S. A. Schuman, editor, *Conference on New Directions in Algorithmic Languages*, pages 157–168, Munich, Germany, August 1975. IFIP Working Group 2.1 on Algol, INRIA.

[8] M. Torgersen. The Expression Problem revisited — four new solutions using generics. In *In Proceedings of the 18th European Conference on Object-Oriented Programming*, pages 123–143. Springer-Verlag, 2004.

[9] P. Wadler. The expression problem. Discussion on the Java Genericity mailing list, Nov 1998. URL homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt.

[10] M. Zenger and M. Odersky. Independently extensible solutions to the expression problem. Technical Report IC/2004/33, EPFL, March 2004. URL http://scala.epfl.ch/docu/related.html.

A. Oliveira and Cook's Code

I stated in Section 3.3 that the Grace code in the body of the paper was based on the Java code of Oliveira and Cook, but simplified. This appendix explains the simplifications and why I made them.

Figure 5 is Oliveira and Cook's original code [6, p.5]. For ease of comparison, Figure 6 is a copy of Figure 1 from page 3 (but with new line numbers). Lines 1–6 of Figure 5 define a type Value used to represent the value of an expression, and two classes that implement it. This is necessary in Java because int and bool have no common supertype. Notice that the implementations of VInt and VBool, although not shown, must fail if an attempt is made to get a value of the wrong type — for example, to getBool from a VInt. Note also that if the set of possible kinds of value is extended, say by adding String-valued expressions, then this code will have to be edited. Writing in Grace, where everything is an Object, it seems to be more natural to define the type Value to be the Grace type Object (Fig 6, line 29). This simplifies the code of the examples at the cost of a slightly poorer diagnostic message if the programmer mistakes one kind of expression for another.

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

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());
}}
```

Figure 5: The part of Oliveira and Cook's code corresponding to Figure 6.

Lines 8–10 define the type Exp, and correspond exactly to line 30 of the Grace code. Lines 11–18 of Figure 5 define the class Lit of integer literals; this corresponds in a fairly obvious way to the Grace factory method lit on lines 32–35 of Figure 6. The convention in Grace is to use lower-case names for classes and capitalized names for types. Methods are public by default in Grace, so there is no need to annotate them; return is implicit at the end of a method, so I omitted the **return** keyword. The Java "constructor" on lines 13–15 of Figure 5 is replaced in Grace by the factory method itself; moreover, there is no need to explicitly define a field x or to assign the parameter i of the factory method to it, because the parameter is itself accessible to the manufactured object. For this reason, it was necessary to choose a different name for the parameter. Type information is optional in Grace, but to parallel the Java code, I have annotated the methods with their return types (shown after the −> symbols).

Similarly, the Java class Add on lines 19–26 corresponds quite closely to the Grace factory method sum on lines 36–40. The Grace code uses two methods l and r to access the operands, rather than public fields.

```
dialect "staticTypes"

type Value = Object
type Exp = { eval −> Value }

factory method lit(i:Number) −> Exp {
    method x −> Number { i }
    method eval −> Value { x }
}
factory method sum(a:Exp, b:Exp) −> Exp {
    method l −> Exp { a }
    method r −> Exp { b }
    method eval −> Value { l.eval + r.eval }
}
// Demonstration:
def threePlusFour:Exp = sum(lit 3, lit 4)
print "{threePlusFour} = {threePlusFour.eval}"
// prints:    an object = 7
```

Figure 6: The Grace module *exp_base* (a copy of Figure 1).