The Expression Problem, Trivially!

Yanlin Wang Bruno C. d. S. Oliveira
The University of Hong Kong, Pokfulam, Hong Kong, China
{ylwang,bruno}@cs.hku.hk

Abstract

This paper presents a novel and simple solution to Wadler's Expression Problem that works in conventional object-oriented languages. Unlike all existing solutions in Java-like languages, this new solution does not use any kind of generics: it relies only on subtyping. The key to the solution is the use of *covariant type refinement* of return types (or fields): a simple feature available in many object-oriented languages, but not as widely known or used as it should be. We believe that our results present valuable insights for researchers and programming language designers interested in extensibility. Furthermore our results have immediate applicability as practical design patterns for improving the extensibility of programs.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages, Patterns

Keywords Expression problem, Object-oriented programming, Design patterns, Modularity, Extensibility

1. Introduction

Solutions to Wadler's Expression Problem (EP) [17] have been a hot topic in programming languages for almost twenty years. Today we know of various solutions to the EP that either rely on *new programming language features* [3, 4, 6, 8], or can be used as *design patterns* [9] in existing languages [12, 13, 15, 16]. The non-triviality of the EP is associated with the fact that existing solutions either require languages with specially crafted type systems, or encodings in existing languages using several techniques to overcome the limitations of the type system. So far, solutions that work in existing languages (such as Java, Scala or Haskell) have employed various techniques and a combination of two different mechanisms: *type-parametrization* and *subtyping*.

This paper shows that conventional subtyping (as found in Scala and Java) is enough to solve Wadler's EP. We present a Scala solution, which is essentially the same code that programmers usually write in a typical (failed) attempt to solve the EP. The only minimal difference is a simple type annotation. The annotation serves the purpose of covariantly refining the extended types. This shows, somewhat surprisingly, that the Wadler's EP can be (almost) trivially solved. We also present a Java solution, which is slightly more involved due to the use of *covariant return types* to simulate covariant type-refinement in fields. Nevertheless, the Java solution is still quite simple and uses no generics either. The code for the solutions presented in this paper is online[1].

We should point out that while the solutions presented here do solve Wadler's original formulation of the EP, they do not immediately scale to harder extensibility problems. For instance the techniques presented here are insufficient to deal with family polymorphism [6] in its full generality. The limitations are discussed in Section 6.

2. Wadler's Expression Problem

The Expression Problem has been widely discussed [5, 11, 14] and was coined by Wadler [17] to illustrate modular extensibility issues in software evolution, especially when involving recursive data structures. Wadler set a simple programming exercise: implementing a language for a very simple form of arithmetic expressions (for example: $1 + 2$ or 3). There is an initial set of features consisting of two types of expressions (integer *literals* and *addition*); and one operation (*evaluation* of expressions). Later, two features, which represent possible evolutions of the original system, are added:

- **New variant:** a new type of expressions, e.g., *subtraction*.
- **New operation:** a new method, e.g., *pretty printing*.

It is usually easy to extend the system in one dimension (either new variants or new operations). In particular, extending the system with new operations in functional languages is easy, but adding new data variants is difficult. While in common object-oriented languages, the dual problem appears:

MODULARITY'16, March 14–17, 2016, Málaga, Spain
© 2016 ACM. 978-1-4503-3995-7/16/03...
http://dx.doi.org/10.1145/2889443.2889448

[1] https://bitbucket.org/yanlinwang/ep_trivially

```
trait Exp { def eval() : Int }
trait Lit extends Exp {
 val x:Int
 def eval() = x
}
trait Add extends Exp {
 val e1, e2 : Exp
 def eval() = e1.eval + e2.eval
}
```

Figure 1. Initial code in the Scala solution.

adding new data variants is easy, but adding new operations is more difficult. Although design patterns like VISITOR pattern [9] allow operations to be added easily, adding new data variants becomes difficult. So the traditional VISITOR pattern does not solve the EP: it merely swaps the dimension of extensibility. The challenge is how to design a programming technique that supports software evolution in both dimensions in a modular way, without modifying the code that has been written previously. The requirements of solutions to the EP are stated more precisely next:

- *Extensibility in both dimensions*: A solution must allow the addition of new data variants and new operations and support extending existing operations.
- *Strong static type safety*: A solution must prevent applying an operation to a data variant which it cannot handle using static checks.
- *No modification or duplication*: Existing code must not be modified nor duplicated.
- *Separate compilation and type-checking*: Safety checks or compilation steps must not be deferred until link or runtime.

There is also a common 5th requirement which is proposed by Zenger and Odersky [18]:

- *Independent extensibility*: It should be possible to combine independently developed extensions, so that they can be used jointly.

3. A Trivial Solution in Scala

This section presents a solution to the EP in Scala. The main Scala feature used here is the support for type refinement of (immutable) *fields*. This simple feature allows us to write the solution to the EP very directly and compactly.

Initial System The initial system shown in Figure 1 defines a trait `Exp` with the evaluation (`eval`) operation. Traits `Lit` and `Add` extend `Exp` with corresponding implementations of `eval`. Note that `e1` and `e2` are immutable member fields, declared as **`val`**s.

Adding a New Variant It is easy to add new data variants to the initial system in Figure 1, while satisfying all the requirements for a solution. For example, trait `Sub` illustrates the addition of new variants, and is almost the same as the definition of trait `Add`.

```
trait ExpP extends Exp { def print():String}
trait LitP extends Lit with ExpP {
 def print() = "" + x
}
trait AddP extends Add with ExpP {
 val e1, e2 : ExpP // type refined!
 def print() = "("+ e1.print + "+" + e2.print +")"
}
```

Figure 2. Adding an operation `print` in the Scala solution.

```
trait Sub extends Exp {
 val e1, e2:Exp
 def eval() = e1.eval - e2.eval
}
```

Adding a New Operation Figure 2 shows an example of extending the initial system with a pretty printing operation. The basic idea is to extend traits `Exp`, `Lit` and `Add` with traits `ExpP`, `LitP` and `AddP`, respectively. Note that the type of member fields `e1` and `e2` in `AddP` is refined! That is, instead of keeping the type of `e1` and `e2` as `Exp`, we change it to a *subtype* (`ExpP`). Changing the type is allowed in Scala because it is just a form of covariant type refinement of types in positive positions, which is well-understood in the theory of object-oriented languages [2].

Importantly, note that it is the lack of this type-refinement that is to blame for typical naive attempts to solve the EP. In a naive attempt, the trait `AddP` would be defined as:

```
// Incorrect: typical code in naive non-solution!
trait AddP extends Add with ExpP {
 // method does not type-check!
 def print() = "("+ e1.print + "+" + e2.print +")"
}
```

The problem is that, because the type of `e1` and `e2` is not refined, the call to the method `print` fails to type-check: the trait `Exp` does not support a `print` method.

Instantiation The Scala solution is easy and concise to use:

```
// Initial system
val l1 = new Lit{val x=4}
val l2 = new Lit{val x=3}
val a = new Add{val e1=l1; val e2=l2}
println("a.eval = " + a.eval)

// Subtraction feature
val s = new Sub{val e1=l1; val e2=l2}
println("s.eval = " + s.eval)

// Print feature
val le1 = new LitP{val x=4}
val le2 = new LitP{val x=3}
val ae = new AddP{val e1=le1; val e2=le2}
println(ae.print + " = " + ae.eval)
```

Here, various objects are created from traits in Figures 1 and 2. The first block of code illustrates how to use the initial system, building a simple expression and evaluating it. The second block shows how to use the subtraction feature. Finally, the

```
trait ExpC extends Exp {
  def collectLit(): List[Int]
}
trait LitC extends Lit with ExpC {
  def collectLit() : List[Int] = x :: List()
}
trait AddC extends Add with ExpC {
  val e1, e2 : ExpC
  def collectLit() : List[Int] = e1.collectLit :::
      e2.collectLit
}
```

Figure 3. Adding an operation `collectLit`.

last block shows how to build expressions and use both pretty printing and evaluation.

4. Independent Extensibility

Systems that satisfy independent extensibility should be able to combine multiple independently developed extensions easily. In this way, programmers can merge several extensions into a single compound one. In a trait-based language like Scala, it is easy to obtain independent extensibility by simply relying on multiple trait-inheritance [18]. To illustrate independent extensibility, we extend the initial system with a new operation `collectLit` (which collects all literal components in an expression) in Figure 3. The code to combine two extensions (with `print` and `collectLit` respectively) is:

```
trait ExpPC extends ExpP with ExpC
trait LitPC extends LitP with LitC with ExpPC
trait AddPC extends AddP with AddC with ExpPC {
  val e1, e2 : ExpPC
}
```

`ExpPC` is the new expression interface supporting `print` and `collectLit` operations; `LitPC` and `AddPC` are the extended variants. Notice that except for the routine of **extend** clauses, we only need to refine the type of `e1,e2` in `AddPC`. Instantiation code is essentially the same as the instantiation code presented in Section 3, thus we omit it here.

5. A Java Solution

This section presents a solution to the EP in Java. Since Java does not support type-refinement of fields, we use *covariant return types* instead to allow refinements of the types of recursive sub-expressions. Figure 4 shows a class diagram summarizing our Java solution.

Initial System The initial system shown in Figure 5 is almost the same as the Scala code presented in Figure 1. The difference is that old member fields `e1` and `e2` in trait `Add` are now replaced by abstract functions `getE1()` and `getE2()`. Therefore the class `Add` becomes an abstract class correspondingly. These abstract *getter* methods will enable future extensions of the initial system to covariantly refine the return types of these methods.

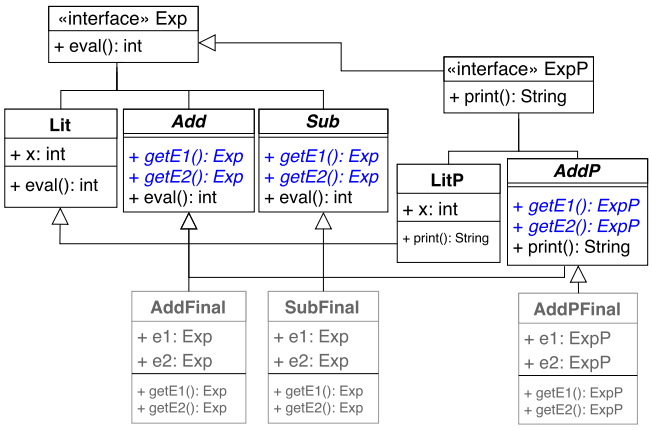

Figure 4. The Java solution overview.

```
interface Exp { int eval(); }
class Lit implements Exp {
   int x;
   Lit(int x) { this.x = x; }
   public int eval() { return x; }
}
abstract class Add implements Exp {
   abstract Exp getE1(); //refinable return type!
   abstract Exp getE2(); //refinable return type!
   public int eval() {
      return getE1().eval() + getE2().eval();
   }
}
```

Figure 5. Initial code in the Java solution.

Adding a New Variant Extending the initial system with a new data variant `Sub` is easy, as shown here:

```
abstract class Sub implements Exp {
   abstract Exp getE1();
   abstract Exp getE2();
   public int eval() {
      return getE1().eval() - getE2().eval();
   }
}
```

Adding a New Operation Figure 6 shows an example of extending the initial system with a new operation `print`. Importantly, note that the definition of the `print()` method in the class `AddP` is well-typed. This is because the types of the getters `getE1()` and `getE2()` are refined, using the covariant return types feature of Java, to return `ExpP` instead of `Exp`. If the types were not refined, then there would be a type-error when using `getE1().print()` or `getE2().print()`, since method `print()` would not be defined in `Exp`.

Instantiation Note that in the initial system, the abstract class `Add` is not immediately usable: abstract classes cannot be directly instantiated. As shown in Figure 7, an additional class `AddFinal` is needed to extend `Add` and provide concrete implementations of abstract methods `getE1(), getE2()` in

```
interface ExpP extends Exp { String print(); }
class LitP extends Lit implements ExpP {
   LitP(int x) { super(x); }
   public String print() { return "" + x; }
}
abstract class AddP extends Add implements ExpP {
   abstract ExpP getE1(); //return type refined!
   abstract ExpP getE2(); //return type refined!
   public String print() {
      return "(" + getE1().print() + " + " +
            getE2().print() + ")";
   }
}
```

Figure 6. Adding an operation `print` in the Java solution.

```
class AddFinal extends Add {
   Exp e1, e2;
   AddFinal(Exp e1, Exp e2) {
      this.e1 = e1;
      this.e2 = e2;
   }
   Exp getE1() { return e1; }
   Exp getE2() { return e2; }
}
```

Figure 7. An additional class for instantiation.

its superclass. With `AddFinal` we can create an expression and execute an operation on it:

```
Exp exp = new AddFinal(new Lit(7), new Lit(4));
System.out.println(exp.eval());
```

Similarly, when updating the system with new operation `print`, an additional class `AddPFinal` is defined for instantiation of the abstract class `AddP` (code for `AddPFinal` is almost the same as `AddFinal`, so we omit it here).

6. Discussion and Limitations

It may seem surprising that such a simple solution to the EP in mainstream languages has not been proposed before in the literature. One possible explanation is that although many languages support covariant type refinement in some form, only Scala allows a very direct solution using type refinement of immutable fields. In Java some more ingenuity (and code) is required to make use of covariant return types. Although the Java solution has some boilerplate (because `Final` classes are needed), that code is mechanical and could be automatically generated. The Java solution can also support independent extensibility by changing classes into interfaces with default methods (supported in Java 8), but for reasons of space we omit that variant here.

Although the idea of covariant refinement has not been applied before to solutions of the EP in mainstream OO languages, it has been a fundamental part of various new language designs aimed at solving extensibility problems. For example, languages that support family polymorphism rely on the fact that family extensions allow covariant type refinements. In languages supporting family polymorphism, it is also possible to have a simple solution to Wadler's EP [7]. One important difference to more conventional type systems like Java is that in family polymorphism covariant type-refinement is also possible for arguments of methods. In contrast Java (or Scala) only allows type-refinements for types used in positive positions (that is, return or field types). There is a good reason for such restriction: it is well-known that naively allowing covariant type-refinement everywhere would lead to type unsoundness. Type systems for family polymorphism need to take special care to ensure that covariant type-refinement can happen everywhere.

Binary and Producer Methods The restriction of type-refinement to types in positive positions implies that binary methods pose extra challenges for extensibility. For example, if expressions were to support a (binary) equality method, then we would want to refine the argument type of the equality method in the extension. However this is not possible in Scala or Java. Producer methods that transform one expression and produce another are possible using the techniques presented here. However, they introduce some code duplication because the original code of the method cannot be reused in extensions. In the original Wadler's EP, the two operations (printing and evaluation) are consumer methods where the recursive type does not occur anywhere in the signature of the method. It is for this special class of methods that our techniques shine and lead to particularly simple solutions.

Mutability Another limitation of the approach presented here is the lack of mutability of the sub-expressions: the Scala solution relies on immutable fields; and the Java solution relies on getters. We do not know how to support mutability using only subtyping. However, if we also allow the use of generics, then we can obtain a variant of the solution presented here that supports mutability and even removes the need for final classes in Java. The idea is to abstract over the type of expressions in classes with sub-expressions. For example, instead of the classes `Add` (in Figure 5) and `AddP` (in Figure 6), the following classes would be used:

```
class Add<E extends Exp> implements Exp {
   E e1, e2;
   public int eval() {return e1.eval() + e2.eval();}
}
class AddP<E extends ExpP> extends Add<E> implements
      ExpP {
   public String print() {
      return e1.print() + " + " + e2.print();
   }
}
```

Now, the fields `e1` and `e2` are mutable, and the types of the fields are refined via the bounds of `E`. This solution can be viewed as a simplification of Torgersen's solution to the EP [16], that avoids uses of F-bounds [1] and excessive type-parametrization. The full source code for this variant, including instantiation code, can be found online. If we go

all the way to Torgersen's solution it is even possible to deal with binary and producer methods. However this comes at the cost of simplicity, as now the code gets filled with numerous type annotations and bounds.

7. Related Work

The EP in Mainstream Languages The solutions to the EP presented here only use subtyping. This is in sharp contrast to existing solutions in various widely-used languages. Essentially all the solutions that we know use a combination of two mechanisms: subtyping and type-parametrization. Wadler proposed a solution using generics in Generic Java to solve the EP. However, he later found a subtle typing problem. Torgersen [16] presents 4 solutions that use a combination of Java generics and subtyping. His solutions require a lot of type parametrization and use advanced features, such as F-bounds [1] or wildcards. Our approaches follow the same structure as Torgersen's first solution. The difference is that covariant return types are used, instead of F-bounded type parameters, to type occurrences of recursive types. Object Algebras [13] are an alternative approach to solve the EP in Java-like languages. While Object Algebras do not require F-bounds or wildcards, they still require generics. There are also solutions to the EP in Haskell (for example [15]), but all of them rely heavily on type-parametrization.

Language-based solutions Various approaches, based on new programming languages or programming language features, can be used to solve the EP. Examples of these include *multi-methods* [3]; *open classes* [4]; *virtual classes* [8]; approaches to *family polymorphism* [6]; and others [10]. The main difference to our work is that those approaches rely on the features targeted at solving extensibility problems, whereas the approach presented here relies only on standard OO language features.

8. Conclusion

This paper presents a new solution to Wadler's Expression Problem using only subtyping. The key idea is to use covariant type refinement to refine recursive types in extensions. This solution does not need any form of type-parametrization.

We believe that the results of our work provide two important insights. Firstly, while it has been widely believed that statically typed functional languages and OOP languages have equal difficulties in solving the Wadler's EP, our work shows that this is not true. Wadler's EP is in fact simpler to solve in OOP languages due to the native support for subtyping. Since traditional functional languages, such as Haskell or ML, have avoided native support for subtyping a similar solution does not directly apply. Secondly, our work shows that, as a benchmark for extensibility, Wadler's EP is perhaps "too easy". The bar can be set higher by requiring, not only consumer methods, but also binary and producer methods (such as a binary equality operation, or an operation that transforms expressions).

Acknowledgments

We thank the reviewers and Erik Walkingshaw for feedback that significantly improved this paper. This work has been sponsored by the Hong Kong Research Grant Council Early Career Scheme project number 27200514.

References

[1] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. C. Mitchell. F-bounded polymorphism for object-oriented programming. In *FPCA '89*, 1989.

[2] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[3] C. Chambers and G. T. Leavens. Typechecking and modules for multimethods. *ACM TOPLAS*, 17:805–843, 1995.

[4] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: modular open classes and symmetric multiple dispatch for java. In *OOPSLA '00*, 2000.

[5] W. R. Cook. Object-oriented programming versus abstract data types. In *FOOL '91*, 1991.

[6] E. Ernst. Family polymorphism. In *ECOOP '01*, 2001.

[7] E. Ernst. The expression problem, Scandinavian style. In *MASPEGHI*, 2004.

[8] E. Ernst, K. Ostermann, and W. R. Cook. A virtual class calculus. In *POPL '06*, 2006.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1995.

[10] J. Hannemann and G. Kiczales. The aspectj web site. http://www.aspectj.org.

[11] S. Krishnamurthi, M. Felleisen, and D. P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *ECOOP '98*. 1998.

[12] B. C. d. S. Oliveira. Modular visitor components. In *ECOOP'09*, 2009.

[13] B. C. d. S. Oliveira and W. R. Cook. Extensibility for the masses: Practical extensibility with object algebras. In *ECOOP'12*, 2012.

[14] J. C. Reynolds. User-defined types and procedural data structures as complementary approaches to type abstraction. In *New Directions in Algorithmic Languages*. 1975.

[15] W. Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423 – 436, 2008.

[16] M. Torgersen. The expression problem revisited – four new solutions using generics. In *ECOOP'04*, 2004.

[17] P. Wadler. The Expression Problem. Email, Nov. 1998. Discussion on the Java Genericity mailing list.

[18] M. Zenger and M. Odersky. Independently extensible solutions to the expression problem. In *FOOL'05*, 2005.