The Expression Problem Revisited

Four new solutions using generics

Mads Torgersen

Computer Science Department
University of Aarhus
Aabogade 34, Aarhus, Denmark
madst@daimi.au.dk

Abstract. The expression problem (aka the extensibility problem) refers to a fundamental dilemma of programming: To which degree 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.
Over the years, many approaches to this problem have been proposed, each having different characteristics of type safety and reusability. While many of these rely on exotic or problem specific language extensions, this paper investigates the solution space within the framework of the soon-to-be mainstream generic extensions of C# and the Java programming language.
Four new solutions are presented which, though quite different, all rely on techniques that can be used in everyday programming.

1 Introduction

A typical structure found in application programs is the one represented by the *Composite* design pattern [1]: a recursive data structure defined by a number of interrelated classes, and a set of operations with specific behaviour for each class. The *expression problem* (or *extensibility problem*) is concerned with the issue of modular extensibility of such structures: Depending on the programming language and the organisation of the code, it is usually straightforward to add either new data types or new operations without changing the original program, but with the price that it is very hard to add the other kind.

There are already many solutions to this problem, all with their own qualities and drawbacks. This paper presents no less than four new solutions, each with its own contributions to the subject area. While many previous approaches are linguistic in nature, introducing special purpose constructs to deal with the problem, the present solutions are all based on the new generic capabilities of C# and the Java programming language [2–5]. Furthermore we establish a terminology framework for assessing the multitude of solutions, using this to compare with a number of previous approaches.

1.1 The expression example

Although the issue has been known for many years, and was discussed by e.g. Reynolds in 1975 [6] and Cook in 1990 [7], the expression problem was named thus by Philip Wadler in 1998 [8] as a pun referring on the one hand to the challenges it poses to the expressive power of programming languages, and on the other to a classical domain in which programming language implementors often encounter it: when representing expressions of a language, along with operations to manipulate them, it is a very real dilemma whether to favor future extension with new expression kinds or new operations over them.

In this paper we shall follow the tradition and use as a running example a processor of a (very!) simple language with the following grammar:

Exp ::= *Lit* | *Add*
Lit ::= *non-negative integer*
Add ::= *Exp* '+' *Exp*

We name the language ALE from the initials of its constituent categories. Along with a representation of the data itself, from the outset the processor is to implement a print() operation to show expressions on screen.

To examine the expression problem we shall then want to try out different representations of ALE in the context of two "future" extensions: with a new operation, eval(), to evaluate expressions, and with a new kind of expression *Neg*:

Exp ::= ... | *Neg*
Neg ::= '−' *Exp*

Of course this is a toy example, limited to the minimum necessary to illustrate the points in the paper. Note that this example differs slightly from the one put forth by Wadler to better highlight problematic issues (recursion in the base case) and weed out orthogonal issues (the handling of return values in visitors).

Given the small amount of application logic involved, it may seem that the amount of infrastructure needed in the examples is overwhelming in comparison. But one should keep in mind that real applications will involve a much larger proportion of application logic. Also, "legacy" client code of a *Composite* structure can also be made reusable by a nondestructive extension, and is therefore likely to constitute another major bulk of code that is spared the need to be hand updated or recompiled.

1.2 The extensibility dilemma

Given, in the manner of the *Composite* design pattern, a number of recursively defined classes describing the data, there are basically two ways of structuring the description of the operations around them, each with extensibility consequences:

Data-centered: Each operation may be defined as a virtual method in the common base class of the data, and overwritten in each specific data class. This is the straightforward object-oriented approach, and has the modular

property that a new class may be easily added without modifications to the existing code. Adding a new operation, however, involves modifying every single data class to implement the data specific behaviour of the new operation.

Operation-centered: Instead the code may be structured according to the *Visitor* design pattern [1]: Each operation is represented by a separate visitor class, containing handler methods (or *visit* methods) for each datatype. All data classes are equipped once and for all with an *accept* method which calls the appropriate visit method of a given visitor with the data object itself as an argument. This structuring makes the data classes robust toward the addition of new operations in the form of new Visitor classes, but unfortunately the addition of a new data class requires all the visitor classes to extend their fixed list of visit methods so that they can deal with the new kind of data.

A straightforward data-centered implementation of ALE may look as in Figure 1. In order to use the code we just have to build instances of the data classes and start calling print() on them.

```
interface Exp {
  void print();
}

class Lit implements Exp {
  public int value;
  Lit(int v) { value = v; }
  public void print() { System.out.print(value); }
}

class Add implements Exp {
  public Exp left, right;
  Add(Exp l, Exp r) { left = l; right = r; }
  public void print() { left.print(); System.out.print('+'); right.print(); }
}
```

Fig. 1. A class-based implementation

As can be seen in Figure 2, an operation-centered implementation requires a bit more setup. To use the implemented functionality, when we have created appropriate instances we have to call their accept() method with an instance of the AlePrint visitor class in order to get the desired output.

Adding a *Neg* expression to the data-centered code calls for a new data class which just has to implement the print() method. This requires no change to the existing code. In the operation-centered implementation, however, all existing visitor classes must be modified to contain a visitNeg() method, breaking the modularity of the extension.

```
interface Exp {
  void accept(AleVisitor v);
}

class Lit implements Exp {
  public int value;
  Lit(int v) { value = v; }
  public void accept(AleVisitor v) { v.visitLit(this); }
}

class Add implements Exp {
  public Exp left, right;
  Add(Exp l, Exp r) { left = l; right = r; }
  public void accept(AleVisitor v) { v.visitAdd(this); }
}

interface AleVisitor {
  void visitLit(Lit lit);
  void visitAdd(Add add);
}

class AlePrint implements AleVisitor {
  public void visitLit(Lit lit) { System.out.print(lit.value); }
  public void visitAdd(Add add) {
    add.left.accept(this); System.out.print('+'); add.right.accept(this);
  }
}
```

Fig. 2. A visitor-based implementation

When adding the eval() operation the situation is reversed. In the visitor-based implementation the operation easily fits in as a new implementation, AleEval, of the AleVisitor interface, but as a virtual eval() method in the data-centered implementation it would have to be added to all classes.

Thus, either choice seems to paint us into a corner, and that is the core of the expression problem. The approach of most solutions is to take one of the above structuring principles as a starting point, and then use various tricks and language constructs to "loosen up" the dilemma. Of the solutions in this paper, the first is fundamentally data-centered, whereas the second and third (which are closely related) are operation-centered. The fourth solution is a hybrid, using both approaches in one implementation.

1.3 A note on code examples

Minimal as the solution examples may be, this is a paper about the structuring of code, and very often the twist undermining a whole approach is hidden in one of those tiny details that one is tempted to gloss over in presentations. Thus, in

the interest of verifiability we have chosen to show the full code for the different solutions.

The solutions are all presented using the familiar syntax of the Java programming language, with the generic enhancements scheduled for the next release (JDK1.5) of the Java platform. The full java source code for the solutions can be accessed at `www.daimi.au.dk/~madst/ecoop04`. In the interest of uniformity, this is the case even for the fourth solution, which in fact does not work *as-is* in a Java setting, where runtime information of generic types is lacking, but would in the generic framework being added to C#. This solution has been hand-translated in a straightforward fashion to obtain a slightly less elegant but still working solution in Java. The solutions have all been compiled and tested using the prototype compiler available for evaluation from Sun Microsystems at `developer.java.sun.com/developer/earlyAccess/adding_generics/index.html`. Any errors are thus due to the formatting process.

1.4 Overview

The following section introduces a terminology framework for characterizing and evaluating solutions to the expression problem. The next four sections (3 to 6) develop and examine as many solutions. Section 7 concludes.

2 Solutions

Many solutions to the expression problem have been proposed over the years. On close scrutiny they differ considerably in their requirements to the language context, as well as in the degree of extensibility they offer, and the limitations they impose. In this section we set up a framework of terminology which can help to assess the characteristics of the many different solutions.

But first of all, let us define exactly what we mean by a "solution":

A solution to the expression problem is a combination of

- *a programming language*
- *an implementation of a Composite structure in that language, and*
- *a discipline for extension*

which allows both new data types and operations to be subsequently added

1. *any number of times*
2. *without modification of existing source code*
3. *without replication of non-trivial code*
4. *without risk of unhandled combinations of data and operations*

This definition is less restrictive than the one given by Philip Wadler in [8]. For one thing, he requires extensions to be made without the need to recompile existing source code, whereas we contend ourselves with the source code not being changed. Thus we include more approaches in the solution space, because some of these are realistic and interesting. In order to maintain a differentiation,

below we classify the different solutions according to their degree of robustness – their "level of extensibility".

Another important difference is that Wadler includes only completely type safe (i.e. cast free) solutions. We have weakened this requirement to state that the solution must handle all combinations.

In his description of the expression problem [9], Kim Bruce goes a bit in the other direction from our definition, being content with "rewriting as little code as possible", whereas we stand firm on "no modifications". In practice, however, his paper is concerned primarily with non-modifying solutions.

2.1 Language context

The expression problem is in fact not specific to an object-oriented setting, but has been discussed also in the context of functional languages. Here, the straightforward solution is to declare an algebraic datatype over the nodes and use type-casing (i.e. pattern matching) functions to express the operations. This is equivalent to the visitor-based solution above, in the sense that it is hard to add new data types modularly, whereas operations are easy. To complete the picture, functional languages may simulate the object-oriented approach using closures, so the same dichotomy of operation versus data extensibility remains.

There are some important differences, however, because functional solutions do not have to deal with statefulness and subclass substitutability. That this can have an impact on the solution is witnessed e.g. by the proposals in [9], many of which depend on a restriction of subclass substitutability for type safety. It is therefore not obvious that solutions from the functional world translate well to an object-oriented counterpart, and in this paper we will look only at proposals dealing with an object-oriented context.

Even so, most approaches depend strongly on the precise set of language constructs available in their environment, and oftentimes new mechanisms are proposed specifically for dealing with the expression problem. Examples of such linguistic approaches include "deep subtyping" [8], "classgroups" [9] and "extensible algebraic datatypes" [10]. Other solutions depend on general-purpose, but not so widespread mechanisms, such as virtual types or multi-methods.

The solutions in this paper are less linguistic. They highlight what can be done with existing or soon-to-be mainstream generics, and so the contributions are in the code structure rather than the mechanisms. Yet, they are still very dependent on the exact constructs available: the third solution makes use of the wildcard mechanism which occurs only in the Java implementation of generics, whereas the fourth solution depends on runtime reification of type variables, which, apart from some research dialects of Java (e.g. NextGen [11] and PolyJ [12]), is available only in the C# version of generics.

2.2 Level of extensibility

A very important characteristic is the degree of reuse of existing code offered by the various approaches. To qualify as a solution we require that extensions fully

reuse nontrivial application logic from the extended code. Thus, there should be no need to duplicate code except for whatever "scaffolding" is needed to set up the extension. We refer to this property as *source-level* extensibility.

As noted above, the definition in [8] requires that existing code need not even be recompiled. This is a much stronger requirement and clearly broadens the applicability of the solution. For instance, recompilation is a big problem in widely distributed code, on which independent third party operations have to be able to interact. We call this degree of reuse *code-level* extensibility. All the solutions in this paper have this property.

Additionally however, it is highly desirable that the data structures themselves, which may be large, persistently stored or part of an application that cannot be allowed to terminate and restart because of the extension, may continue to be usable after the extension has taken place. Thus, we want the objects created before the extension to survive and remain compatible afterward. We call this property *object-level* extensibility. The third and fourth of our solutions in this paper have that property.

2.3 Generative programming

Recent years have seen a flourishing of so-called *generative* approaches to programming, in which higher level source code is used to direct the generation or manipulation of base level source code. A popular example is aspect-oriented programming, which e.g. allows extra methods to be added to a class from a separate unit of source code. This clearly solves the expression problem (in our definition), but leads to an under-the-hood mangling of the original class, that requires recompilation to a binary format.

In general, any discipline of destructive modification of a source code text can be automated, and thus turned into a generative approach. Of course an automated approach may be more safe than hand-editing, since it can be made to check that various invariants are maintained.

While generative programming seems to have its uses, it does however suffer from the need to deploy new binary code for every extension. Due to the focus of this paper on a generic Java and C# context, generative approaches are out of reach, and will not be further treated.

2.4 Basic Approach

Most proposals take as a starting point either a data-centered approach, making it hard to add new operations, or an operation-centered (visitor-based) approach, making it equally hard to add new data types. The choice may depend on various language specific issues, but one thing is certain: If object-level extensibility is desired, then the data-centered approach must be ruled out. This is because the addition of operations will then require new virtual methods to be added to all the classes representing data, e.g. by the use of subclassing. Old data objects created from the unmodified classes will not have the new methods, and will therefore not support the new operations.

With the operation-centered approach, however, there is hope that existing objects can be made to work with newly added or extended visitors. Of course, the converse restriction exists, that old visitor objects cannot survive the addition of new data types, because they have no appropriate visit method. But this problem is less grave because visitors represent operations, which we do not expect to be lasting, just as we do not expect to reuse the activation record of a method call.

Our first solution examines the data-centered approach, whereas the second is operation-centered. The third solution enhances the second in a manner that allows for object level extensibility with some drawbacks, whereas the fourth solution ventures a new *hybrid* approach which allows both data components *and* visitor objects to be reused across extensions.

2.5 Extension graph

Most solutions assume that extensions happen linearly, one after the other, and that a given extension knows about the previous version. One might well imagine, however, that a given piece of software is extended independently by two different parties, and that one might later want to merge them.

The approach in [10] relies heavily on a linear extension discipline, because dispatching method calls are chained all the way up through the extension path. There it is argued that a merge is rarely needed in practice, and that it can be handled by other means.

Our extension approaches rely on subclassing, and a merge of multiple extensions would require the inheritance mechanism to also be multiple. Thus the Java versions in this paper are restricted to linearity, but in other languages or dialects with e.g. mixins [13, 14] they might not be.

2.6 Surrounding code

An important but often overlooked aspect of solutions is the effect they have on the reusability of the surrounding code. There are two main sorts of external code that depend on the implementation of the *Composite* structure itself:

Creation code: The part of the application that is in charge of actually producing instances of the datatype classes. If a solution relies on defining new data types for each extension, then new constructors need to be called.

Client code: The code that calls the operations on the datatype classes. If the operations are represented by visitors, which get subclassed by new extensions, then we have the dual situation to the one described for creation code: we need to be able to create the right kinds of visitors.

Furthermore, both types of external code will have to be type parameterised, if the classes they depend on are. Due to space limitations the descriptions in the following sections on how to support reuse of surrounding code will not show the full code for doing so. The full code which has been used to test

these approaches includes factories and similar infrastructure, as well as simple creation and client methods which are reused across extensions, in order to demonstrate the extensibility of surrounding code. As mentioned in Section 1 this Java code can be downloaded from `www.daimi.au.dk/~madst/ecoop04`.

2.7 Type safety

As noted above, Wadler requires full type safety of solutions. The reality is, however, that a number of approaches make use of type casts, but still provide other interesting perspectives on the problem. For instance, neither of [15, 10] is fully statically type safe, yet we wish to include those for comparison, rather than branding them as irrelevant *á priori* . To minimise the danger, both of these approaches propose language extensions handled by preprocessing, to ensure that casts are inserted using a safe discipline.

While the first three of our solutions in this paper are statically typed, the fourth one resorts to runtime type checks in exchange for other benefits. The casts occur only in an initial dispatching framework, so the extenders will not have to write unsafe code themselves. They will however have to stick to a linear extension discipline without the aid of a type checker to avoid "holes" of unhandled combinations of operations and data.

3 A data-centered approach

We may note that the data-centered version of ALE in Figure 1 is considerably simpler than the visitor-based one in Figure 2, and perhaps more appealing from an object-oriented point of view. In this section we therefore investigate what it takes to equip it for code-level extensibility, and what the major challenges are.

Adding new kinds of data does not pose an immediate problem in this setting; what we have to deal with is how to add a new virtual method to all members of the type hierarchy. In our specific example, we wish to add an eval() method to the Exp interface and the classes Lit and Add. We can achieve this nondestructively by introducing a new interface EvalExp extending the Exp interface with an eval() method. New versions of Lit and Add must then extend the old ones while implementing the new interface. This will lead to e.g. a new class EvalAdd of the following form:

```
class EvalAdd extends Add implements EvalExp {
  public int eval() { return left.eval()+right.eval(); }
}
```

Immediately we are in trouble: the inherited instance variables left and right are of type Exp, and therefore do not have an eval() method to call recursively. Thus, the compiler fails to type check the above code.

As also pointed out in [9], we can attempt to address this with genericity: we may parameterise expressions with the type of their children. While Add will allow its type parameter to be any kind of Exp, EvalAdd may then restrict it to be

a kind of EvalExp, thus ensuring that its children also have an eval() method. In both cases, the children must have the same child type as their parents, which we arrange by making use of *F-bounds* [16] in the declarations of the type variables i.e., the type variable occurs in its own bound. Figure 3 shows the parameterised code for ALE.

```
interface Exp⟨C extends Exp⟨C⟩⟩ {
  void print();
}

class Lit⟨C extends Exp⟨C⟩⟩ implements Exp⟨C⟩ {
  public int value;
  Lit(int v) { value = v; }
  public void print() { System.out.print(value); }
}

class Add⟨C extends Exp⟨C⟩⟩ implements Exp⟨C⟩ {
  public C left, right;
  Add(C l, C r) { left = l; right = r; }
  public void print() { left.print(); System.out.print('+'); right.print(); }
}
```

Fig. 3. A data-centered implementation of ALE with code-level extensibility

The extension with an eval() operation can now be undertaken in a type safe manner: in Figure 4, EvalAdd not only adds an eval() method implementation, but also expects a child type parameter that extends EvalExp.

```
interface EvalExp⟨C extends EvalExp⟨C⟩⟩ extends Exp⟨C⟩ {
  int eval();
}

class EvalLit⟨C extends EvalExp⟨C⟩⟩ extends Lit⟨C⟩ implements EvalExp⟨C⟩ {
  EvalLit(int v) { super (v); }
  public int eval() { return value; }
}
class EvalAdd⟨C extends EvalExp⟨C⟩⟩ extends Add⟨C⟩ implements EvalExp⟨C⟩ {
  EvalAdd(C l, C r) { super (l,r); }
  public int eval() { return left.eval()+right.eval(); }
}
```

Fig. 4. An extension of Figure 3 with eval() methods

Thus, a type safe extension has been achieved. The type bookkeeping is overwhelming, however, and it can be hard to spot the few lines of actual application logic. But there is one more twist: because all the classes are F-bounded, there does not at this point exist any class or interface that can be used as a type parameter for them.

In a manner typical of programming with F-bounds, we first have to create a new set of non-generic subclasses that *fix* the F-bound on themselves. This has to be done for every layer of extension that needs to be used in actual application code, since there is no other way of getting to construct instances of the classes. For the evaluation extension, the fixing classes can be seen in Figure 5.

```
interface EvalExpFix extends EvalExp⟨EvalExpFix⟩ {}
class EvalLitFix extends EvalLit⟨EvalExpFix⟩ implements EvalExpFix {
  EvalLitFix(int v) { super (v); }
}
class EvalAddFix extends EvalAdd⟨EvalExpFix⟩ implements EvalExpFix {
  EvalAddFix(EvalExpFix l, EvalExpFix r) { super (l,r); }
}
```

Fig. 5. Fixed point classes for the F-bounds in Figure 4

The classes have trivial bodies and add no new semantics to our application, but simply "tie down" the F-bounds.

Finally, then, we have a set of eval()-enabled classes that can be instantiated in application code like this:

```
EvalExpFix e1 = new EvalLitFix(2);
EvalExpFix e2 = new EvalLitFix(3);
EvalExpFix e3 = new EvalAddFix(e1,e2);
e3.print(); System.out.println(" = " + e3.eval());
```

This is a type-safe data-centered code-level-extensible solution to the expression problem, and indeed the first in the literature to use only standard generics. Yet, the reader will have noticed that the initial simplicity of the data-centered approach has disappeared.

3.1 Surrounding code

Using the techniques of the *Abstract Factory* design pattern, we can limit the complexity of object creation in this approach to the extension code itself, and keep it out of surrounding creation code. The following method, taken from the on-line code examples for this paper, demonstrates the point:

```
static ⟨C extends Exp⟨C⟩⟩ C build(AleFactory⟨C⟩ f) {
  return f.makeAdd(f.makeLit(2),f.makeLit(3));
}
```

The method builds a specific expression tree without heed to the operations its nodes contain. It is reusable across extensions because it is parameterised over the two kinds of things that extensions alter: The base expression type and the creation procedure for its instances.

Of course this means more work when extending with new data classes such as Neg, because a new NaleFactory interface extending AleFactory has to introduce a method for creating new Neg instances of the right kind.

Client code is made reusable simply by parameterizing it over the base expression type, as in:

```
static ⟨C extends Exp⟨C⟩⟩ void show(C exp) { exp.print(); }
```

3.2 Related work

This solution is quite similar in structure to the solution by Bruce [9] using self types as the type of child nodes. This works only because of the rather peculiar semantics of self types in that work, where the self type in a class is not the class itself, but its nearest interface. Thus, the self type of Add would not be Add but Exp.

Compared to F-bounds, the problem with self types is that they cannot be fixed, and therefore may require type-exact references (i.e., no subsumption) to work properly.

On the other hand, F-bounds cannot fully express self types. We shall see an example of this shortcoming in the following section. Thus, independently of the expression problem, the lack of true self types in Java and C# leads to a less permissive type system in some situations.

4 An operation-centered approach

Given the unexpected complexity of the data-centered solution, we now turn to the other, initially more complex viewpoint of the operation-centered approach. Starting from the visitor-based version of the ALE code in Figure 2, the hard problem now is to extend the language to NALE by adding an additional datatype Neg representing negation.

The argument runs somewhat dual to the data-centered approach: If we add the Neg class, we need to extend our visitors to handle it, and we can do so nondestructively only by subclassing:

```
interface NaleVisitor extends AleVisitor {
  void visitNeg(Neg neg);
}
class NalePrint extends AlePrint implements NaleVisitor {
  public void visitNeg(Neg neg) {
    System.out.print('−'); neg.exp.accept(this);
  }
class Neg implements Exp {
  public Exp exp;
  public void accept(AleVisitor v) { v.visitNeg(this); } //Type error!!!
}
```

Again we are in type checking trouble: this time the culprit is the accept() method of Neg, which expects an AleVisitor rather than a NaleVisitor. It has to, in order to implement the accept() method of Exp, but this means that v is not known to have a visitNeg() method.

Again we may resort to type parameterization of the datatype hierarchy in order to allow the accept() method of Neg to expect more of its visitor than its fellow expression types. Note however, that this time the data classes are not parameterised with themselves, which was what lead to all the hassle with F-bounds in the previous solution, but rather with the kind of Visitor they accept().

In the visitor classes, the visit methods must adjust to the fact that the classes they visit are now parameterised. We can obtain this by parameterizing the visit methods themselves over the visitor type of their argument expressions. For Add and its corresponding visit method in AlePrint, we would expect something like this to work:

```
class Add⟨V extends AleVisitor⟩ implements Exp⟨V⟩ {
  public Exp⟨V⟩ left, right;
  public void accept(V v) { v.visitAdd(this); }
}
class AlePrint implements AleVisitor {
  ...;
  public ⟨V extends AleVisitor⟩ void visitAdd(Add⟨V⟩ add) {
    add.left.accept(this); //Type error!!!
    System.out.print('+');
    add.right.accept(this); //Type error!!!
  }
}
```

Surprisingly, however, this does not type check. Because now the AlePrint visitor does not know that the *children* of add will actually accept() it. All we know from the signature of the visitAdd() method is that they will accept V's, but we do not know at this point in the code that **this** is a V.

It *is*, though. Because of the structure of the *Visitor* pattern, visitAdd() is invoked only from a single point in the whole program, and that is in the accept()

method of Add. There, we *do* know that the visitor called is one that both the Add object itself *and* its children are able to accept(), simply because the children by definition accept the same visitor type V as their parent. So how can we pass this knowledge on to the visitAdd() method?

No amount of F-bounded self-parameterization of the visitor classes will help us here. It is a well known shortcoming of F-bounds that they cannot express true self types. To the best of our knowledge there is no way in the generic type system of the forthcoming Java release to get '**this**' in the visitAdd() method typed as a V. This is where [9] has to give up, and the solution presented by Wadler in [8] was later found to be unsafe because of this same issue.

There is a trick, however, which, considering the pain caused by this hurdle, is provocatively simple. We observe that, although the visitAdd() method of AlePrint needs a V object to work on, it does not need to know that this object is indeed **this** object. We may therefore simply add as an extra argument to visitAdd() the visitor to use for recursive calls, and then let the accept() method of Add pass the visitor object *to itself* in the call of visitAdd(). The full ALE implementation with this approach is shown in Figure 6.

Despite its simplicity, this trick is one of the major contributions of this paper, because it finally moves the visitor-based approach to the expression problem into the realm of static type safety.

Finally we now have the scaffolding needed to extend the set of expressions with a Neg class, using the same trick once more for the visitNeg() method. See Figure 7 for the full code.

4.1 Surrounding code

Creation code is easily made reusable across extensions by parameterizing it with the Visitor kind of its products and then using the constructors directly, as in:

```
static ⟨V extends AleVisitor⟩ Exp⟨V⟩ build() {
  return new Add⟨V⟩(new Lit⟨V⟩(2), new Lit⟨V⟩(3));
}
```

Reusable client code can be obtained by using a variation of the *Abstract Factory* pattern to abstract over the creation of appropriate visitors for handling the operations.

4.2 Related work

A special case of client code is the operations themselves, when calling the same or other operations recursively on substructures of the data. In our simple examples we have assumed that there is only a need to call recursively with the same visitor object. Krishnamurthi et.al. [15] propose dealing with the more general situation by using factory methods [1], unfortunately in a type-unsafe manner.

```
interface Exp⟨V extends AleVisitor⟩ {
  void accept(V v);
}

class Lit⟨V extends AleVisitor⟩ implements Exp⟨V⟩ {
  public int value;
  Lit(int v) { value = v; }
  public void accept(V v) { v.visitLit(this); }
}

class Add⟨V extends AleVisitor⟩ implements Exp⟨V⟩ {
  public Exp⟨V⟩ left, right;
  Add(Exp⟨V⟩ l, Exp⟨V⟩ r) { left = l; right = r; }
  public void accept(V v) { v.visitAdd(this, v); }
}

interface AleVisitor {
  ⟨V extends AleVisitor⟩ void visitLit(Lit⟨V⟩ lit);
  ⟨V extends AleVisitor⟩ void visitAdd(Add⟨V⟩ add, V self);
}

class AlePrint implements AleVisitor {
  public ⟨V extends AleVisitor⟩ void visitLit(Lit⟨V⟩ lit) {
    System.out.print(lit.value);
  }
  public ⟨V extends AleVisitor⟩ void visitAdd(Add⟨V⟩ add, V self) {
    add.left.accept(self); System.out.print('+'); add.right.accept(self);
  }
}
```

Fig. 6. An operation-based implementation of ALE with code-level extensibility

```
class Neg⟨V extends NaleVisitor⟩ implements Exp⟨V⟩ {
  public Exp⟨V⟩ exp;
  Neg(Exp⟨V⟩ e) { exp = e; }
  public void accept(V v) { v.visitNeg(this, v); }
}

interface NaleVisitor extends AleVisitor {
  ⟨V extends NaleVisitor⟩ void visitNeg(Neg⟨V⟩ neg, V self);
}

class NalePrint extends AlePrint implements NaleVisitor {
  public ⟨V extends NaleVisitor⟩ void visitNeg(Neg⟨V⟩ neg, V self) {
    System.out.print("-("); neg.exp.accept(self); System.out.print(")");
  }
}
```

Fig. 7. An extension of Figure 6 adding a Neg class

5 Object-level extensibility using wildcards

The solution above effectively divides the node classes of the different phases into separate, incompatible families, each characterised, in the form of a type parameter, by the specific brand of Visitor they are capable of accepting. Wildcards, a new mechanism to appear with generics in the forthcoming version of the Java platform (JDK1.5/Tiger), aim at allowing the programmer to bridge the gap between such separate families of classes [17]. Wildcards (represented by question marks) abstract over different instantiations of a given parameterised class, guided by given bounds on the type parameters. For instance

```
Exp⟨? super NaleVisitor⟩ e;
```

declares a variable e which may contain objects of type Exp⟨*T*⟩ for *any* *T* that is a supertype of NaleVisitor. Thus, it may for example be assigned an instance of Add⟨AleVisitor⟩. To maintain static type safety, the type rules for wildcards impose certain restrictions on method calls, but none of these apply in the given situation.

To obtain object-level extensibility, we should be able to continue to use old ALE expression trees after an extension to NALE has occurred. Already, NaleVisitors can be accepted by instances of Exp⟨AleVisitor⟩, simply because of subclassing. However, we would like for the old trees also to be usable as subtrees of new composite expressions, e.g., of Add⟨NaleVisitor⟩. We can obtain this by using wildcards in the types of children:

```
class Add⟨V extends AleVisitor⟩ implements Exp⟨V⟩ {
  public Exp⟨? super V⟩ left, right;
  public void accept(V v) { v.visitAdd(this, v); }
}
```

Defining composite nodes like this, subtrees with a more general visitor type than V are allowed. As a consequence, not only can we reuse old trees, but we also lift the requirement on leaf nodes to have a type parameter they do not use, only for the purpose of being allowed into the company of like-parameterised classes. Thus, with a few minor changes, in Figure 8 we have modified the solution of the previous section to provide object-level extensibility. Figure 9 shows the modified Neg extension. Only the Neg class itself has changed, the visitors are as in Figure 7.

There is a price to pay for this change, though: With the wildcard type on child nodes, we lose the exact knowledge about the child type of the children themselves – the "grandchild type" as it were. Thus, it is effectively disallowed to write into them anything other than expressions of the original unextended framework, in this case Exp⟨AleVisitor⟩s. This means that we cannot in general write visitors that *change* the structure of a given tree. If we could, we might accidentally store new NALE nodes inside old ALE trees, and any code still using old AleVisitors would be prone to a runtime type error when traversing it.

```
interface Exp⟨V extends AleVisitor⟩ {
  void accept(V v);
}

class Lit implements Exp⟨AleVisitor⟩ {
  public int value;
  Lit(int v) { value = v; }
  public void accept(AleVisitor v) { v.visitLit(this); }
}

class Add⟨V extends AleVisitor⟩ implements Exp⟨V⟩ {
  public Exp⟨? super V⟩ left, right;
  Add(Exp⟨? super V⟩ l, Exp⟨? super V⟩ r) { left = l; right = r; }
  public void accept(V v) { v.visitAdd(this, v); }
}

interface AleVisitor {
  void visitLit(Lit lit);
  ⟨V extends AleVisitor⟩ void visitAdd(Add⟨V⟩ add, V self);
}

class AlePrint implements AleVisitor {
  public void visitLit(Lit lit) { System.out.print(lit.value); }
  public ⟨V extends AleVisitor⟩ void visitAdd(Add⟨V⟩ add, V self) {
    add.left.accept(self); System.out.print('+'); add.right.accept(self);
  }
}
```

Fig. 8. A modification of Figure 6 using wildcards to obtain object-level extensibility

Depending on the type of application, this limitation may or may not be a problem. In compilers for programming languages, it is rare to see a destructive update of the structure of syntax trees. However, in a small step evaluator for e.g., lambda expressions, tree manipulation is at the core of the semantics, and this restriction would be a real showstopper.

5.1 Surrounding code

The surrounding code is where we reap the benefits of this approach: code written at a certain time may keep global references to expressions around, which stay valid even across extensions. Old code will continue to work, and new code is sure not to violate the integrity of the old expression objects. Thus extension is a much less destructive endeavour than in the previous approaches.

6 A hybrid approach

We now turn to a completely different approach. The solution presented here is based on the following idea: if using either a data-centered or an operation-

```
class Neg⟨V extends NaleVisitor⟩ implements Exp⟨V⟩ {
  public Exp⟨? super V⟩ exp;
  Neg(Exp⟨? super V⟩ e) { exp = e; }
  public void accept(V v) { v.visitNeg(this, v); }
}
```

Fig. 9. An extension of Figure 8 adding a Neg class. Visitors are identical to Figure 7 and have been omitted

centered approach leads to so much difficulty, can we find a sweet spot in between?

More precisely, one might view the extensibility problems with both approaches as deriving from the need to "backpatch" existing code when new is added. Virtual methods avoid backpatching of operations when data types are added, whereas visitors avoid backpatching of data types when operations are added. So why not combine these approaches to use virtual methods when adding data types, *and* use visitors when adding operations?

This would require any given operation to be represented both as a virtual method and a visitor. When a new operation is added, it provides functionality for existing data types by means of a visitor, but it also defines a new specialised expression interface with the operation added as a method. Subsequently added data types must then implement this interface, and furthermore provide a new specialised visitor interface with visit methods for the new data types.

This approach is attractive because both data and visitor instances remain valid across new extensions. Compared to the previous solution, which used wildcards to soften the boundaries between the type families originating from the different phases of extension, in this approach there *are* no separate families. As a result there is no need for type parameterization of child types of composite nodes, wherefore the limitations of the wildcard approach on modification of the tree structure are gone.

This sounds almost too good to be true, and sure enough there is a major catch: we cannot implement it without casts. The hard part turns out to be the dispatch of operation calls. Given an expression and a visitor object, we need to figure out whether the operation:

- should be called as a method on the expression,
- should be called as a visit method on the visitor, or
- neither apply and we need to call a default method on the visitor.

We have been able to think of nothing better than to use type tests (using instanceof in Java) for this dispatch. To stop the type unsafe code from spilling out all over the extensions, we parameterise visitors with the kind of data types having the corresponding operation as a method, and we parameterise data types with the kind of visitors that have visit methods for them. In this way we can implement the type testing once and for all in a framework of base classes, as can be seen in Figure 10.

```
interface Exp {
  void handle(Visitor v);
}

interface Visitor {
  void apply(Exp e);
  void default(Exp e);
}

abstract class Node⟨V extends Visitor⟩ implements Exp {
  public final void handle(Visitor v) {
    if (v instanceof V) accept((V)v);
    else v.default(this);
  }
  abstract void accept(V v);
}

abstract class Op⟨E extends Exp⟩ implements Visitor {
  public final void apply(Exp e) {
    if (e instanceof E) call((E)e);
    else e.handle(this);
  }
  abstract void call(E e);
  public void default(Exp e) {
    throw new IllegalArgumentException("Expression problem occurred!");
  }
}
```

Fig. 10. A dispatch framework for a hybrid solution with full object-level extensibility

The dispatch starts with the apply() method of visitors. It is implemented in the abstract Op class (which all concrete visitors will extend) to call the proper method on its argument expression, if it exists, or otherwise ask the expression to do the dispatch using the handle() method. This in turn will check if the visitor has a proper visit method, or otherwise call the default() method of the visitor, which, by default, throws an exception.

Figure 11 shows how ALE can be implemented in this framework. The expression classes implement the print() operation using the class-based approach, but also have accept() methods to deal with future additions. The Print visitor, on the other hand, is trivial, since no expression will ever need a visit() method on it. An AleVisitor interface is provided for future extension.

In Figure 12 an eval() operation is added. It provides visit methods for the already existing data classes, and an interface EvalExp for future extension.

The addition of a Neg class in Figure 13 is quite symmetric to this. Methods are implemented for existing operations, and a NaleVisitor defined for future extension.

```
interface PrintExp extends Exp {
  void print(Print print);
}

class Print extends Op⟨PrintExp⟩ implements Visitor {
  void call(PrintExp e) { e.print(this); }
}

class Lit extends Node⟨AleVisitor⟩ implements PrintExp {
  public int value;
  Lit(int v) { value = v; }
  public void print(Print print) { System.out.print(value); }
  void accept(AleVisitor v) { v.visitLit(this); }
}

class Add extends Node⟨AleVisitor⟩ implements PrintExp {
  public Exp left, right;
  Add(Exp l, Exp r) { left = l; right = r; }
  public void print(Print print) {
    print.apply(left); System.out.print('+'); print.apply(right);
  }
  void accept(AleVisitor v) { v.visitAdd(this); }
}

interface AleVisitor extends Visitor {
  void visitLit(Lit lit);
  void visitAdd(Add add);
}
```

Fig. 11. A hybrid implementation of ALE based on the framework of Figure 10

```
class Eval extends Op⟨EvalExp⟩ implements AleVisitor {
  int result;
  public final int eval(Exp e) { apply(e); return result; }
  void call(EvalExp e) { result = e.eval(this); }
  public void visitLit(Lit lit) { result = lit.value; }
  public void visitAdd(Add add) { result = eval(add.left) + eval(add.right); }
}

interface EvalExp extends PrintExp {
  int eval(Eval eval);
}
```

Fig. 12. An extension of Figure 11 with an eval() operation

```
class Neg extends Node⟨NaleVisitor⟩ implements EvalExp {
  public Exp exp;
  Neg(Exp e) { exp = e; }
  public void print(Print print) {
    System.out.print("−("); print.apply(exp); System.out.print(")");
}
  public int eval(Eval eval) { return −eval.eval(exp); }
  void accept(NaleVisitor v) { v.visitNeg(this); }
}

interface NaleVisitor extends AleVisitor {
  void visitNeg(Neg neg);
}
```

Fig. 13. A further extension of Figure 12 with a Neg class

In the forthcoming release of the Java platform, it is not possible to use casts and the instanceof operation on type variables. This is one of the few essential differences between the generic frameworks of C# and the Java programming language, and comes down to differences in implementation techniques. In this case, however, the limitation in Java is not grave. We can simulate the two operations by adding the following methods to the Node class:

```
abstract boolean isV(Visitor v);
abstract V asV(Visitor v);
```

and the following methods to the Op class:

```
abstract boolean isE(Exp e);
abstract E asE(Exp e);
```

Both must be implemented in the subsequent concrete specialisations of the two classes, when the type parameters are fixed to specific classes on which the runtime type operations work. This is similar to the automated translation performed by NextGen [11] to make up for these shortcomings of generic Java.

6.1 Surrounding code

Reuse of surrounding code in this approach is almost for free. Creation code just instantiates the data classes directly. Client code may directly instantiate operation objects, and apply() them to expression objects. Both data and visitor objects may be kept around as global data, and will still work when new data and operation types are added.

In summary, the previous three approaches allow the *extender* of the data types and operations the safety of a type checked extension discipline, while putting a certain burden of parameterization on the *user* of these structures,

along with limitations on object reuse. The present solution requires more discipline of the extender, who must make sure to always extend the latest previous version, without the aid of the type checker. But in return it completely liberates the user from the worries of the expression problem, essentially allowing any usage patterns of the naive non-extensible approaches of Figure 1 and 2.

6.2 Related work

Odersky and Zenger [10] make use of *defaults* to handle the situations that "fall through" due to the lack of static safety, so that operations are called on data for which they are not defined. It is argued that this approach is useful in the setting they have investigated, an extensible compiler for Java-like languages. However, it is not obvious that there will always be a sensible default action for unhandled nodes in other domains, and throwing a runtime exception explicitly is hardly better than getting a runtime type error.

This approach is reminiscent of what may be achieved with *multimethods* (see e.g. [18]), which dispatch at runtime to the most specific implementation fitting their arguments. If a most-general fallback implementation is supplied, no runtime type errors can occur, as argued by Castagna in the context of a related problem [19]. However, the practical problem of providing meaningful such defaults remains.

In our solution above, the type checker ensures that defaults will not need to be called unless the extender has broken the discipline of linear extension, so the situation is somewhat less severe.

7 Conclusion

The scene of mainstream object-oriented programming languages is ever evolving, and genericity in the form of parameterised classes is becoming a standard component of most popular platforms. This calls for a re-evaluation of old problems and apparent dichotomies in the new context.

In this paper we have shown by examples that standard parametric genericity is a powerful means to obtain type safe solutions. Furthermore we have shown that the features peculiar to the two main language contenders in the market, namely Java's wildcards and C#'s reified type parameters, both have a real impact in terms of extensibility.

A number of novel concrete techniques have been proposed, which are constructive and general, and may be readily applied in real world settings. Thus, they can have a great impact on the extensibility and safety of future applications making use of the *Composite* structure.

A "full and final" solution to expression problem would combine the complete and straightforward object-level extensibility of our hybrid solution with full static type safety. This is still an important challenge for the future, and is unlikely to be achieved only with the linguistic means employed in this paper. Yet we hope that this paper has brought us closer to this ideal, and that it will inspire others to pursue it.

8 Acknowledgments

Thanks are due to Gilad Bracha, Philip Wadler and the anonymous reviewers for helpful comments and corrections.

References

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Abstraction and Reuse of Object-Oriented Designs. Addison-Wesley (1994)
2. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification – Third Edition. Third. edn. Addison-Wesley (2004)
3. Bracha, G., Odersky, M., Stoutamire, D., Wadler, P.: Making the future safe for the past: Adding genericity to the java programming language. [20]
4. ECMA: C# language specification. `http://www.ecma-international.org/publications/standards/Ecma-334.htm` (2002)
5. Kennedy, A., Syme, D.: Design and implementation of generics for the .NET common language runtime. In Norris, C., Fenwick, J.J.B., eds.: Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01). Volume 36.5 of ACM SIGPLAN Notices., N.Y., ACMPress (2001) 1–12
6. Reynolds, J.C.: User-defined types and procedural data as complementary approaches to data abstraction. In Schuman, S.A., ed.: New Directions in Algorthmic Languages. INRIA (1975) Reprinted in D. Gries, ed, Programming Methodology, Springer-Verlag, 1978 and in C. A. Gunter and J. C. Mitchell, eds, Theoretical Aspects of Object-Oriented Programmig, MIT Press, 1994.
7. Cook, W.R.: Object-oriented programming versus abstract data types. In: REX Workshop on Foundations of Object-oriented Languages. Number 489 in LNCS, Springer-Verlag (1990)
8. Wadler, P.: The expression problem. Posted on the Java Genericity mailing list (1998)
9. Bruce, K.B.: Some challenging typing issues in object-oriented languages. In Bono, V., Bugliesi, M., eds.: Electronic Notes in Theoretical Computer Science. Volume 82., Elsevier (2003)
10. Zenger, M., Odersky, M.: Extensible algebraic datatypes with defaults. In: Proceedings of the International Conference on Functional Programming. (2001)
11. Cartwright, R., Steele, G.L.: Compatible genericity with runtime-types for the Java programming language. [20]
12. Myers, A., Bank, J., Liskov, B.: Parameterized types for Java. In: Conf. Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Paris, France, POPL97, ACM Press. Neil D. Jones, editor. (1997)
13. Bracha, G., Cook, W.: Mixin-based inheritance. In: Object Oriented Programming: Systems, Languages and Applications/European Conference on Object-Oriented Programming, Ottawa, Canada, OOPSLA/ECOOP90, ACM Press. Norman K. Meyrowitz, editor. (1990) 303–311
14. Flatt, M., Krishnamurthi, S., Felleisen, M.: Classes and mixins. In: Principles of Programming Languages, San Diego, California, POPL98, ACM Press. David MacQueen, editor. (1998) 171–183

15. Krishnamurthi, S., Felleisen, M., Friedman, D.P.: Synthesizing object-oriented and functional design to promote re-use. In: European Conference on Object-Oriented Programming, Brussels, Belgium, ECOOP98, LNCS 1445, Springer Verlag. Eric Jul, editor. (1998) 91–113
16. Canning, P., Cook, W., Hill, W., Olthoff, W., Mitchell, J.: F-bounded polymorphism for object-oriented programming. In: ACM Conference on Functional Programming and Computer Architecture, ACM Press (1990) 273–280
17. Torgersen, M., Hansen, C.P., Ernst, E., von der Ahé, P., Bracha, G., Gafter, N.: Adding wildcards to the java programming language. In: Proceedings of the ACM Symposium of Applied Computing. (2004)
18. Bobrow, D.G., DeMichiel, L.G., Gabriel, R.P., Keene, S.E., Kiczales, G., Moon, D.A.: Common lisp object system specification. ACM Sigplan Notices, special issue **23** (1988)
19. Castagna, G.: Covariance and contravariance: Conflict without a cause. ACM Transactions on Programming Languages and Systems **17** (1995) 431–447
20. Object Oriented Programming: Systems, Languages and Applications, Vancouver, BC, ACM Press. Craig Chambers, editor. (1998)