Independently Extensible Solutions to the Expression Problem

Matthias Zenger, Martin Odersky

École Polytechnique Fédérale de Lausanne
INR Ecublens
1015 Lausanne, Switzerland

Technical Report IC/2004/33

Abstract

The *expression problem* is fundamental for the development of extensible software. Many (partial) solutions to this important problem have been proposed in the past. None of these approaches solves the problem of using different, independent extensions jointly. This paper proposes solutions to the expression problem that make it possible to combine independent extensions in a flexible, modular, and type-safe way. The solutions, formulated in the programming language SCALA, are affected with only a small implementation overhead and are easy to implement by hand.

1 The Expression Problem

Since software evolves over time, it is essential for software systems to be extensible. But the development of extensible software poses many design and implementation problems, especially, if extensions cannot be anticipated. The *expression problem* is probably the most fundamental one among these problems. It arises when recursively defined datatypes and operations on these types have to be extended simultaneously. The term *expression problem* was originally coined by Phil Wadler in a post on the *Java-Genericity* mailing list [25], in which he also proposed a solution written in an extended version of GENERIC JAVA [3]. Only later it appeared that Wadler's solution could not be typed.

For this paper, we paraphrase the problem in the following way: Suppose we have a datatype which is defined by a set of cases and we have processors which operate on this datatype. There are primarily two directions along which we can extend such a system:

- The extension of the datatype with new data variants,
- The addition of new processors.

We require that processors handle only a finite number of data variants and thus do not provide *defaults* which could handle arbitrary cases of future extensions. The challenge is now to find an implementation technique which satisfies the following list of requirements:

- *Extensibility in both dimensions:* It should be possible to add new data variants and adapt existing operations accordingly. Furthermore, it should be possible to introduce new processors.
- *Strong static type safety:* It should be impossible to apply a processor to a data variant which it cannot handle.
- *No modification or duplication*: Existing code should neither be modified nor duplicated.
- *Separate compilation:* Compiling datatype extensions or adding new processors should not encompass re-type-checking the original datatype or existing processors.

We add to this list the following criterion:

- *Independent extensibility:* It should be possible to combine independently developed extensions so that they can be used jointly [21].

Implementation techniques which meet the last criterion allow systems to be extended in a *non-linear* fashion. Such techniques typically allow programmers to consolidate independent extensions in a single compound extension as illustrated by Figure 1. By contrast, without support for independent extensibility, parallel extensions diverge, even if they are completely orthogonal [7]. This makes a joint use of different extensions in a single system impossible.

This paper presents two families of new solutions to the expression problem. One family is based on object-oriented decomposition while the other is based on functional decomposition using the visitor pattern. In its original form, each of these decomposition techniques allows extensibility only in one direction (data or operations), yet disallows extensibility in the other. The solutions presented here achieve independent extensibility of data and operation extensions. They are sufficiently simple and concise to be immediately usable by programmers.

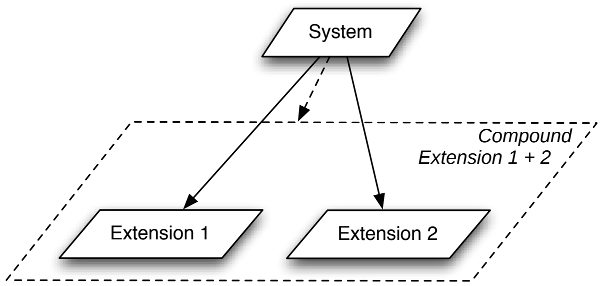

Figure 1: Combination of independent extensions.

Our solutions are expressed in the programming language SCALA [16]. SCALA is a strongly statically typed programming language which fuses object-oriented and functional programming concepts. For instance, (SML-style) module systems are expressed in a purely object-oriented way by identifying modules with objects, functors with classes, and signatures with interfaces. It follows from this identification that objects in SCALA can contain types as members. Furthermore, these type members can be either abstract or concrete. The *path-dependent types* of the *νObj* calculus [17] give a type theoretic foundation for languages like SCALA where types can be members of objects.

In module systems, abstract type members are primarily used for information hiding — they allow one to abstract from concrete implementations. In this paper they are used as a means of composition. We will see that each decomposition technique uses an abstract type member to keep the system open for future extensions in the "dual" dimension (i.e. the dimension in which extensions are normally not possible).

Two other type-systematic constructs explored in *νObj* and implemented in SCALA also play important roles in our solutions. *Mixin composition* allows to merge independent extensions. *Explicitly typed self references* overcome a problem in the visitor-based solutions which made Wadler's original proposals untypable.

SCALA has been designed to interact smoothly with JAVA or .NET host environments. All solutions in this paper compile as given with the current SCALA compiler [16] and can be executed on a Java VM, version JDK 1.4 or later.

The rest of the paper is organized as follows. Section 2 analyzes previous work on the expression problem based on the criteria mentioned initially. Section 3 discusses an independently extensible solution to the expression problem formulated in an object-oriented programming style. An alternative approach based on a functional decomposition is presented in Section 4. Section 5 discusses the implemenation of binary methods. Section 6 concludes with an analysis of the language features that are required by the discussed approaches.

2 Partial Solutions

The expression problem has been intensively studied in the literature. However, none of the proposed solutions satisfies all the requirements stated in Section 1. This section gives an overview over some of the most important solutions proposed in the past.

Object-oriented decomposition In object-oriented languages, the *Interpreter* design pattern [11] can be used to implement datatypes in an extensible fashion. Here, a datatype would be implemented by an abstract superclass which specifies the signature of methods that implement the various processors. Concrete subclasses represent the data variants and implement the processors. This approach makes it easy to add new data variants simply by defining new subclasses, but adding new processors involves modifications of the abstract superclass as well as all concrete subclasses.

Functional decomposition With the *Visitor* design pattern [11] it is possible to address the problem in a more functional fashion. This pattern allows one to separate the representation of data from functionality operating on such data. Processors are encapsulated in Visitor objects which provide for every data variant a method that handles the particular case. This approach makes it straightforward to write new processors, but adding new data variants requires that all existing processors are modified to include methods that handle the new cases.

Extensible visitors Krishnamurti, Felleisen, and Friedman propose the *Extensible Visitor* pattern [13], a slightly modified variant of the Visitor design pattern which makes it possible to add both new data variants and new processors. Unfortunately, this approach is based on type casts which circumvent the type system and therefore make extensions unsafe. In this pattern, all existing visitor classes have to be subclassed whenever a new variant class is added. Otherwise a runtime error will appear as soon as an old visitor is applied to a new variant.

Extensible visitors with defaults Zenger and Odersky refine the *Extensible Visitor* pattern into a programming protocol in which datatype extensions do not automatically entail adaptations of all existing processors and vice versa [26, 27]. Technically, extensibility of data and functionality is achieved by adding default cases to type and visitor definitions; these default cases handle all possible future extensions. While this approach allows programmers to reuse existing visitors for new data variants and therefore does not suffer from the runtime errors described above, it is still not fully satisfactory, since it allows to apply visitors to data variants for which the visitor was not designed for originally.

Multi-methods Programming languages supporting multiple dispatch via multi-methods provide good support for extensibility with default cases. MultiJava [8] is a JAVA-based programming language that allows programmers to add new methods to existing classes without modifying existing code and without breaking encapsulation properties. While new, externally specified methods require default cases, internal methods (i.e. methods that are defined inside of the corresponding class) are not subject to this restriction. A precise analysis of the constraints that are required to enable modular typechecking for such internal and external methods is given by Millstein, Bleckner, and Chambers, in their work on EML [15]. Opposed to all the approaches mentioned before, EML makes it possible to use independent extensions jointly.

Generic visitors Palsberg and Jay's *Generic Visitors*, also called *Walkabouts*, offer a way to completely decouple data representations from function definitions [19]. Therefore, walkabouts are very flexible to use and to extend. But since they rely on reflective capabilities of the underlying system, this approach lacks static type-safety and is subject to substantial runtime penalties. Grothoff recently showed that the performance decrease can be avoided by using runtime code generation techniques [12].

Self types Recently, Bruce presented a way to make the Interpreter design pattern extensible [4]. His approach is based on the existence of a new **ThisType** type construct, referring to the public interface of the self reference **this** inside of a class. Like **this**, the meaning of **ThisType** changes when a method whose signature refers to **ThisType** is inherited in a subclass. This feature makes it possible to keep the type of the data variants open for future extensions. A severe limitation of this approach is that for type-safety reasons, the exact runtime type of the receiver of a method referring to **ThisType** has to be known at compile-time. A further limitation is that **ThisType** cannot be used to make the visitor design pattern extensible.

Generic classes Solutions to the expression problem which rely on generic classes and F-bounds have recently been proposed by Torgersen [23]. Similar to our approach, Torgersen proposes two kinds of solutions: one *data-centered* solution based on an object-oriented decomposition, and a *operation-centered* solution based on a functional decomposition using the visitor design pattern. Torgersen's solutions satisfy our first four requirements stated in Section 1, but do not address the problem of independent extensibility. Another drawback is the relatively extensive and complex programming protocol the programmer has to observe. For instance, his data-centered solution requires a fixed point operation for all classes at each instantiation, which makes it cumbersome to use the schema in practice. His operation-centered solution relies on a clever trick to pass a visitor object as argument to itself in order to overcome the typing problems encountered by Wadler. However, this is not exactly an obvious technique for most programmers and it becomes progressively more expensive in the case of several mutually recursive visitor classes. An interesting variation of Torgersen's solution uses JAVA's wildcards [24] to achieve *object-level extensibility*, i.e. reusability of actual expression objects across extensions.

3 Object-Oriented Decomposition

This section presents a solution of the expression problem in SCALA using an object-oriented approach. Following Wadler's original problem statement, we evolve a simple datatype for representing arithmetic expressions together with operations on this type by incrementally adding new datatype variants and new operations.

3.1 Framework

We start with a single data variant `Num` for representing integer numbers and an operation `eval` for evaluating expressions. An object-oriented implementation is given in the following program:

```
trait Base {
  type exp <: Exp;
  trait Exp {
    def eval: int
  }
  class Num(v: int) extends Exp {
    val value = v;
    def eval = value
  }
}
```

The *trait* `Exp` lists the signature of all available operations and thus defines an interface for all data variants. Traits in SCALA are very similar to interfaces in JAVA; the main difference is that traits may contain concrete implementations for some methods.

The only data variant is implemented by class `Num`. This class extends `Exp` with a method `value` which returns the corresponding integer value. It also defines a concrete implementation for operation `eval`.

To keep the set of operations on expressions open for future extensions, we abstract over the expression type and use an *abstract type* `exp` whenever we want to refer to expression objects. An abstract type definition introduces a new named type whose concrete identity is unknown; type bounds may be used to narrow possible concrete incarnations of this type. This mechanism is used in the program above to declare that `exp` is a subtype of our preliminary expression interface `Exp`.

Since we want to be able to refer to our three abstractions `exp`, `Exp`, and `Num` as a whole, we wrap them into a top-level trait `Base`. `Base` has to be subclassed in order to either extend it, or to use it for a concrete application. The latter is illustrated in the following program:

```
object BaseTest extends Base with Application {
  type exp = Exp;
  val e: exp = new Num(7);
  Console.println(e.eval);
}
```

This program defines a top-level *singleton object* whose class is an extension of trait `Base`. The *type alias* definition `type exp = Exp` overrides the corresponding abstract type definition in the superclass `Base`, turning the abstract type `exp` into a concrete one (whose identity is `Exp`). The last two lines in the code above instantiate the `Num` class and invoke the `eval` method. The clause `with Application` in the header of the object definition is a *mixin class composition* [2] which, in this case, adds a `main` method to `BaseTest` to make it executable. We will explain mixin class compositions in the next subsection.

3.2 Data Extensions

Linear Extensions The object-oriented decomposition scheme makes it easy to create new data variants. In the following program we present two extensions of trait `Base`. `BasePlus` extends our system by adding a new `Plus` variant, `BaseNeg` defines a new `Neg` variant. Note that in general, we type expressions using the abstract type `exp` instead of the type defined by the concrete class `Exp`.

```
trait BasePlus extends Base {
  class Plus(l: exp, r: exp) extends Exp {
    val left = l; val right = r;
    def eval = left.eval + right.eval
  }
}
trait BaseNeg extends Base {
  class Neg(t: exp) extends Exp {
    val term = t;
    def eval = - term.eval;
  }
}
```

Combining Independent Extensions We can now deploy the two extensions independently of each other; but SCALA also allows us to merge the two independent extensions into a single compound extension. This is done using a *mixin class composition* mechanism which includes the member definitions of one class into another class. The following line will create a system with both `Plus` and `Neg` data variants:

```
trait BasePlusNeg extends BasePlus with BaseNeg;
```

Trait `BasePlusNeg` extends `BasePlus` and incorporates all the member definitions of trait `BaseNeg`. Thus, it inherits all members from trait `BasePlus` and all the new members defined in trait `BaseNeg`. Note that the members defined in trait `Base` are not inherited twice. The mixin class composition with trait `BaseNeg` only incorporates the new class members and omits the ones that get inherited from `BaseNeg`'s superclass `Base`.

Mixin class composition in SCALA is similar to both the mixin construct of Bracha [2] and to the trait composition mechanism of Schärli, Ducasse, Nierstrasz, and Black [20]. As opposed to multiple inheritance, base classes are inherited only once. In a mixin composition `A with B with C`, class `A` acts as actual superclass of mixins `B` and `C`, replacing the declared superclasses of `B` and `C`. To maintain type soundness, `A` must be a subclass of the declared superclasses of `B` and `C`. A `super` reference in either `B` or `C` will refer to a member of class `A`. As is the case for trait composition, SCALA's mixin composition is commutative in the mixins — `A with B with C` is equivalent to `A with C with B`.

A class inheriting from `A with B with C` inherits members from all three base classes. Concrete members in either base class replace abstract members with the same name in other base classes. Concrete members of the mixin classes `B` and `C` always replace members with the same name in the superclass `A`. If some concrete member `m` is implemented in both `B` and `C`, then the inheriting class has to resolve the conflict by giving an explicit overriding definition of `m`.

Unlike the original mixin and trait proposals, SCALA does not have different syntactic constructs for classes on the one hand and mixins or traits on the other hand. Every class can be inherited as either superclass or mixin base class. Traits in SCALA are simply special classes without state or constructors. This distinction is necessary because of the principle that base classes are inherited only once. If both `B` and `C` have a base class `T`, then the two instances are unified in the composition `A with B with C`. This presents no problem as long as `T` is a trait, i.e. it is stateless and does not have an explicit constructor. For non-trait base classes `T`, the above mixin composition is statically illegal. The idea to have a common syntactic construct for classes and mixins/traits is due to Bracha [1].

3.3 Operation Extensions

Adding new operations requires more work than adding new data variants. For instance, here is how we can add a `show` method to expressions of our base language.

```
trait Show extends Base {
  type exp <: Exp;
  trait Exp extends super.Exp {
    def show: String;
  }
  class Num(v: int) extends super.Num(v) with Exp {
    def show = value.toString();
  }
}
```

In this example, we first have to create an extended trait `Exp` which specifies the new signature of all operations (the old ones get inherited from the old `Exp` trait, the new ones are specified explicitly), then we have to subclass all data variants and include implementations of the new operations in the subclasses. Furthermore, we have to narrow the bound of our abstract type exp to our newly defined Exp trait. Only this step makes the new operations accessible to clients since they type expressions with the abstract type exp.

Note that the newly defined Exp and Num classes shadow the former definitions of these classes in superclass Base. The former definitions are still accessible in the context of trait Show via the **super** keyword.

Shadowing vs. overriding constitutes one of the key differences between classes in SCALA and virtual classes [14]. With virtual classes, class members override equally named class members of a base class, whereas in SCALA the two class members exist side by side (similar to what happens to object fields in JAVA or C#). The overriding behavior of virtual classes is potentially quite powerful, but poses type safety problems due to covariant overriding. There exist proposals to address the type safety problems of virtual classes [22, 10], but the resulting type systems tend to be complicated and have not yet been explored fully.

Linear extensions We can adapt our previously defined systems so that even data variants defined in extensions of Base support the show method. Again, this is done with a mixin class composition. This time we mix the new Show trait into extensions of existing traits such as BasePlusNeg of Section 3.2. Since all our data variants have to support the new show method, we have to create subclasses of the inherited data variants which support the new Exp trait.

```
trait ShowPlusNeg extends BasePlusNeg with Show {
  class Plus(l: exp, r: exp) extends super.Plus(l, r)
                                  with Exp {
    def show = left.show + "+" + right.show;
  }
  class Neg(t: exp) extends super.Neg(t) with Exp {
    def show = "-(" + term.show + ")";
  }
}
object ShowPlusNegTest extends ShowPlusNeg
                          with Application {
  type exp = Exp;
  val e: exp = new Neg(
                 new Plus(new Num(7), new Num(6)))
  Console.println(e.show + "␣=␣" + e.eval);
}
```

The previous program also illustrates how to use the new system. The singleton object ShowPlusNegTest first closes the (still open) definition of type exp, then it instantiates an expression involving all different kinds of data variants. Finally, both the eval and the show method are invoked.

Tree transformer extensions So far, all our operations took elements of the tree only as their receiver arguments. We now show what is involved when writing *tree transformer* operations, which also return tree elements as results. As an example, let's add a method dble to the expression type defined in trait BasePlusNeg. Method dble is supposed to return a new expression which evaluates to a number which is twice the value of the original expression.

Instead of first introducing the new operation in the base system (which would also be possible), we choose to specify it directly in an extension. The following program illustrates the steps required to add method dble to the expression type defined in trait BasePlusNeg.

```
trait DblePlusNeg extends BasePlusNeg {
  type exp <: Exp;
  trait Exp extends super.Exp {
    def dble: exp;
  }
  def Num(v: int): exp;
  def Plus(l: exp, r: exp): exp;
  def Neg(t: exp): exp;
  class Num(v: int) extends super.Num(v) with Exp {
    def dble = Num(v * 2);
  }
  class Plus(l: exp, r: exp)
              extends super.Plus(l, r) with Exp {
    def dble = Plus(left.dble, right.dble);
  }
  class Neg(t: exp) extends super.Neg(t) with Exp {
    def dble = Neg(t.dble);
  }
}
```

Note that we cannot simply invoke the constructors of the various expression classes in the bodies of the dble methods. This is because method dble returns a value of type exp, the type representing extensible expressions, but all data variant types like Plus and Num extend only trait Exp which is a supertype of exp. We can establish the necessary relationship between exp and Exp only at the stage when we turn the abstract type into a concrete one (with the type alias definition **type** exp = Exp). Only then, Num is also a subtype of exp. Since the implementation of dble requires the creation of new expressions of type exp, we make use of abstract factory methods, one for each data variant. The concrete factory methods are implemented at the point where the abstract type exp is resolved. For instance, they can be implemented at the point where we use the new dble method:

```
object DblePlusNegTest extends DblePlusNeg
                          with Application {
  type exp = Exp;
  def Num(v: int): exp = new Num(v);
  def Plus(l: exp, r: exp): exp = new Plus(l, r);
  def Neg(t: exp): exp = new Neg(t);
  val e: exp = Plus(Neg(Plus(Num(1), Num(2))),
                    Num(3));
  Console.println(e.dble.eval);
}
```

All examples presented here are type-safe, in the sense that it is impossible to mix data from different languages, nor to invoke an operation on a data object which does not understand it. For instance, here is what happens when we try to compile a program which violates both requirements.

```
object erroneous {
  val t1 = new ShowPlusNegTest.Num(1);
  val t2 = new DblePlusNegTest.Neg(t1);
//                              ^
// type mismatch;
// found   : ShowPlusNegTest.Num
// required: DblePlusNegTest.Exp

  val t3 = t1.dble;
//           ^
// value dble is not a member of
// ShowPlusNegTest.Num
}
```

Combining independent extensions Finally we show how to combine the two traits `ShowPlusNeg` and `DblePlusNeg` to obtain a system which provides expressions with both a `double` and a `show` method. In order to do this, we have to perform a *deep mixin composition* of the two traits; i.e. we have to combine the two top-level traits `ShowPlusNeg` and `DblePlusNeg` as well as the traits and classes defined inside of these two top-level traits. Since SCALA does not provide a language mechanism for performing such a deep mixin composition operation, we have to do this by hand, as the following program demonstrates:

```
trait ShowDblePlusNeg extends ShowPlusNeg
                      with DblePlusNeg {
  type exp <: Exp;
  trait Exp extends super[ShowPlusNeg].Exp
            with super[DblePlusNeg].Exp;
  class Num(v: int)
    extends super[ShowPlusNeg].Num(v)
       with super[DblePlusNeg].Num(v)
       with Exp;
  class Plus(l: exp, r: exp)
    extends super[ShowPlusNeg].Plus(l, r)
       with super[DblePlusNeg].Plus(l, r)
       with Exp;
  class Neg(t: exp)
    extends super[ShowPlusNeg].Neg(t)
       with super[DblePlusNeg].Neg(t)
       with Exp;
}
```

For merging the two `Exp` traits defined in `ShowPlusNeg` and `DblePlusNeg`, we extend one of the two traits and mix the other trait definition in. We use the syntactic form **`super`**`[...]` to specify to which concrete `Exp` trait we are actually referring. The same technique is used for the other three classes `Num`, `Plus`, and `Neg`.

The previous examples show that the object-oriented approach described in this section supports both data and operation extensions and provides good support for combining independent extensions on demand. While combining extensions with new data variants is relatively simple to implement, combining extensions with different new operations is technically more difficult.

4 Functional Decomposition

For applications where the data type implementations are fixed and new operations are added frequently, it is often recommended to use the *Visitor* design pattern. This pattern physically decouples operations from data representations. It provides a double dispatch mechanism to apply externally defined operations to data objects. In this section we will show how to use a techniques similar to the ones presented in the previous section to implement this pattern in an extensible fashion, allowing both data and operation extensions and combinations thereof.

4.1 Framework

The following program presents a framework for a visitor-based implementation of expressions supporting an `eval` operation. In this framework, we use the type defined by trait `Exp` directly for representing expressions. Concrete expression classes like `Num` implement the `Exp` trait which defines a single method `accept`. This method allows programmers to apply a visitor object to the expression. A visitor object is an encoding for an operation. It provides methods of the form `visit...` for the various expression classes. The `accept` method of a concrete expression class simply selects its corresponding `visit` method of the given visitor object and applies it to its encapsulated data.

```
trait Base {
  trait Exp {
    def accept(v: visitor): unit;
  }
  class Num(value: int) extends Exp {
    def accept(v: visitor): unit = v.visitNum(value);
  }
  type visitor <: Visitor;
  trait Visitor {
    def visitNum(value: int): unit;
  }
  class Eval: visitor extends Visitor {
    var result: int = _;
    def apply(t: Exp): int = { t.accept(this); result }
    def visitNum(value: int): unit = {
      result = value;
    }
  }
}
```

To keep the set of expression classes open, we have to abstract over the concrete visitor type. We do this with the abstract type `visitor`. Concrete implementations of the visitor interface such as class `Eval` typically implement its bound `Visitor`.

Class Eval uses a variable result for returning values. This is necessary since the visitNum method has as result type unit, and therefore cannot return a nontrivial result. It would seem more natural to return a result directly from the visit methods. Then the Visitor class would have to be parameterized with the type of the results. However, in that case the abstract type name visitor would be bounded by the *type constructor* Visitor. Such abstract type constructors have not yet been studied in detail in the context of *vObj* and consequently have not been implemented in SCALA.

To facilitate the processing of result values in clients, the Eval class provides instead an apply method which returns the most recent result value. The body of this method exhibits a technical problem. We have to call t.accept(**this**), but the type Eval is not a subtype of the abstract visitor type visitor required by the accept method of expressions. In SCALA we can overcome this problem by declaring the type of **this** explicitly. Such an *explicitly typed self reference* is expressed in the program above with the statement :visitor directly following the name of class Eval. The type assigned to **this** is arbitrary; however, classes with explicitly typed self references can only be instantiated if the type defined by the class is a subtype of the type assigned to **this**. Since Eval is not a subtype of visitor we cannot create instances of Eval in the context of the top-level trait Base. For creating new instances of Eval we would have to resort to factory methods.

Note that explicitly typed self references are different from Bruce's mytype construct [6], even though the two techniques address some of the same problems. Unlike mytype, explicitly typed self references do not change covariantly with inheritance. Therefore, they are a good fit with standard subtyping, whereas mytype is a good fit with matching [5].

4.2 Data Extensions

Linear extensions New data variants are added to the system by including new visit methods into the Visitor trait and by overriding the abstract type visitor with the extended Visitor trait. The next program extends Base by adding a new Plus expression class.

```
trait BasePlus extends Base {
  type visitor <: Visitor;
  trait Visitor extends super.Visitor {
    def visitPlus(left: Exp, right: Exp): unit;
  }
  class Plus(left: Exp, right: Exp) extends Exp {
    def accept(v: visitor): unit =
      v.visitPlus(left, right);
  }
  class Eval: visitor extends super.Eval with Visitor {
    def visitPlus(l: Exp, r: Exp): unit = {
      result = apply(l) + apply(r);
    }
  }
}
```

The top-level trait BasePlus also defines a new Eval class implementing the refined Visitor trait which can also handle Plus objects. Note that we have to annotate the new Eval class again with an explicit type for its self reference. This is required because for type-safety reasons class extensions have to redefine self types covariantly.

In the same way, we can now create another extension BaseNeg which adds support for negations.

```
trait BaseNeg extends Base {
  type visitor <: Visitor;
  trait Visitor extends super.Visitor {
    def visitNeg(term: Exp): unit;
  }
  class Neg(term: Exp) extends Exp {
    def accept(visitor: v): unit =
      visitor.visitNeg(term);
  }
  class Eval: visitor extends super.Eval with Visitor {
    def visitNeg(term: Exp): unit = {
      result = -apply(term);
    }
  }
}
```

Combining independent extensions We now compose the two independent extensions BasePlus and BaseNeg such that we have a system providing both, addition and negation expressions. In the previous object-oriented decomposition scheme such a combination was achieved using a simple mixin composition. In the functional approach, a deep mixin composition is required to achieve the same effect:

```
trait BasePlusNeg extends BasePlus with BaseNeg {
  type visitor <: Visitor;
  trait Visitor extends super.Visitor
                with super[BaseNeg].Visitor;
  class Eval: visitor extends super.Eval
                          with super[BaseNeg].Eval
                          with Visitor;
}
```

The program extends the previous extensions BasePlus and mixes in the other extension BaseNeg. All concrete visitor implementations such as Eval are also merged by mixin composing their implementations in the two base classes. The SCALA type system [17] requires that abstract types such as visitor are refined covariantly. Since the bounds of visitor in the two previous extensions are not compatible, we have to explicitly override the abstract type definition of visitor such that the new bound is a subtype of both old bounds. Above, this is done by creating a new Visitor trait that merges the two previous implementations.

The following implementation shows how to use a language. As usual, the scheme is the same for base language and extensions. In every case, we close the operations under consideration by fixing the visitor type with a type alias.

```
object BasePlusNegTest extends BasePlusNeg {
  type visitor = Visitor;
  val op: visitor = new Eval;
  Console.println(op.apply(
    new Plus(new Num(1), new Neg(new Num(2)))));
}
```

4.3 Operation Extensions

Adding new operations to a visitor-based system is straightforward, since new operations are implemented simply with new classes implementing the visitor interface. The following code shows how to add a new operation `Dble` to the `BasePlusNeg` system. The `Dble` operation returns an expression representing the double value of a given expression.

```
trait DblePlusNeg extends BasePlusNeg {
  class Dble: visitor extends Visitor {
    var result: Exp = _;
    def apply(t: Exp): Exp = {
      t.accept(this); result
    }
    def visitNum(value: int): unit = {
      result = new Num(2 * value)
    }
    def visitPlus(l: Exp, r: Exp): unit = {
      result = new Plus(apply(l), apply(r))
    }
    def visitNeg(term: Exp): unit = {
      result = new Neg(apply(term))
    }
  }
}
```

In a similar fashion we can create a second, independent extension `ShowPlusNeg` which adds an operation for displaying expressions in textual form.

```
trait ShowPlusNeg extends BasePlusNeg {
  class Show: visitor extends Visitor {
    var result: String = _;
    def apply(t: Exp): String = {
      t.accept(this); result
    }
    def visitNum(value: int): unit = {
      result = value.toString()
    }
    def visitPlus(l: Exp, r: Exp): unit = {
      result = apply(left) + "+" + apply(right)
    }
    def visitNeg(term: Exp): unit = {
      result = "-(" + apply(term) + ")"
    }
  }
}
```

Combining Independent Extensions We can now implement a system which supports both operations `Dble` and `Show` by using a simple shallow mixin class composition involving the two orthogonal independent extensions `DblePlusNeg` and `ShowPlusNeg`:

```
trait ShowDblePlusNeg extends DblePlusNeg
                            with ShowPlusNeg;
```

This example illustrates a duality between functional and object-oriented approaches when it comes to combining independent extensions. The functional decomposition approach requires a deep mixin composition for merging data extensions but only a shallow mixin composition for merging operation extensions. For the object-oriented approach, the situation is reversed; data extensions can be merged using shallow mixin composition whereas operation extensions require deep mixin composition.

Hence, the fundamental strengths and weaknesses of both decomposition approaches still show up in our setting, albeit in a milder form. A merge of extensions in a given dimension which was impossible before now becomes possible, but at a higher cost than a merge in the other dimension.

5 Binary Methods

The previous examples discussed operations where the tree appeared as receiver or as method result. We now study *binary methods*, where trees also appear as a non-receiver arguments of methods. As an example, consider adding a structural equality test `eql` to the expression language. `x eql y` should evaluate to **`true`** if `x` and `y` are structurally equal trees. The implementation given here is based on object-oriented decomposition; the dual implementation based on functional decomposition is left as an exercise for the reader. We start with an implementation of the `eql` operation in the base language.

```
trait Equals extends Base {
  type exp <: Exp;
  trait Exp extends super.Exp {
    def eql(other: exp): boolean;
    def isNum(v: int) = false;
  }
  class Num(v: int) extends super.Num(v) with Exp {
    def eql(other: exp): boolean = other.isNum(v);
    override def isNum(v: int) = v == value;
  }
}
```

The idea is to implement `eql` using double dispatch. A call to `eql` is forwarded to a test method which is specific to the receiver type. For the `Num` class this test method is `isNum(v: int)`. A default implementation of `isNum` which always returns **`false`** is given in class `Exp`. This implementation is overridden in class `Num`.

5.1 Data Extensions

An extension with additional data types requires additional test methods which are analogous to `isNum`. Hence, we need to use a combination of our schemes for data and operation extensions. Here is an extension of class `Equals` with `Plus` and `Neg` types.

```
trait EqualsPlusNeg extends BasePlusNeg with Equals {
  type exp <: Exp;
  trait Exp extends super[BasePlusNeg].Exp
            with super[Equals].Exp {
    def isPlus(l: exp, r: exp): boolean = false;
    def isNeg(t: exp): boolean = false;
  }
  class Num(v: int) extends super[Equals].Num(v)
                    with Exp;
  class Plus(l: exp, r: exp) extends Exp
                             with super.Plus(l, r) {
    def eql(other: exp): boolean = other.isPlus(l, r);
    override def isPlus(l: exp, r: exp) =
      (left eql l) && (right eql r)
  }
  class Neg(t: exp) extends Exp
                    with super.Neg(t) {
    def eql(other: exp): boolean = other.isNeg(t);
    override def isNeg(t: exp) = term eql t
  }
}
```

This extension adds test methods of the form `isPlus(l: exp, r: exp)` and `isNeg(t: exp)` to class `Exp`. Since the addition of these test methods constitutes an operation extension, we need to refine the abstract type `exp`, similar to what was done in Section 3.3.

Note that SCALA allows any binary method to be used as an infix operator. An expression such as `left eql l` is syntactic sugar for `left.eql(l)`.

Note also that the order of inheritance is reversed in classes `Plus` and `Neg` when compared to class `Num`. This is due to the restriction that the superclass `A` in a mixin composition `A with B` must be a subclass of the declared superclass of the mixin `B`. In our example, `Num`'s superclass is `Num` as given in `Equals`, which is a subclass of class `Exp` as given in `Equals`. On the other hand, the superclass of `Plus` is the current definition of `Exp`, which is a subclass of `Exp` as given in `BasePlusNeg`. The difference in the inheritance order is due to the fact that classes `Num` and `Plus`/`Neg` themselves come from different base classes of `EqualsPlusNeg`. `Num` comes from class `Equals` whereas `Plus` and `Neg` come from class `BasePlusNeg`.

5.2 Operation Extensions

A desirable property of binary methods is that they adapt automatically to (operation) extensions. This property holds in our setting, as is demonstrated by the following example, which adds the `show` method to the classes in trait `EqualsPlusNeg` by mixin-composing them with the contents of class `ShowPlusNeg` from Section 3.3.

```
trait EqualsShowPlusNeg extends EqualsPlusNeg
                        with ShowPlusNeg {
  type exp <: Exp;
  trait Exp extends super[EqualsPlusNeg].Exp
            with super[ShowPlusNeg].Exp;
  class Num(v: int)
    extends super[EqualsPlusNeg].Num(v)
       with super[ShowPlusNeg].Num(v)
       with Exp;
  class Plus(l: exp, r: exp)
    extends super[EqualsPlusNeg].Plus(l, r)
       with super[ShowPlusNeg].Plus(l, r)
       with Exp;
  class Neg(term: exp)
    extends super[EqualsPlusNeg].Neg(term)
       with super[ShowPlusNeg].Neg(term)
       with Exp;
}
```

As can be seen from this example, we apply precisely the deep mixin composition scheme for merging operation extensions — compare with trait `ShowDblePlusNeg` in Section 3.3. This shows that no special techniques are needed to adapt binary methods to operation extensions.

We conclude with a main program which uses the `eql` and `show` methods. Again, no special provisions are needed for binary methods.

```
object EqualsShowPlusNegTest extends EqualsPlusNeg
                             with Application {
  type exp = Exp;
  val term1 = new Plus(new Num(1), new Num(2));
  val term2 = new Plus(new Num(1), new Num(2));
  val term3 = new Neg(new Num(2));
  Console.print(term1.show + "=" + term2.show + "?␣");
  Console.println(term1 eql term2);
  Console.print(term1.show + "=" + term3.show + "?␣");
  Console.println(term1 eql term3);
}
```

6 Discussion

We have presented two families of type-safe solutions to the expression problem, which are dual to each other. One family is based on object-oriented decomposition, the other on functional decomposition using the visitor pattern. Either family makes it easy to extend a system in one dimension — data extensions for object-oriented decomposition and operation extensions for functional composition. Extensions in the dual dimension are made possible by abstracting over a type — the tree type in the case of object-oriented decomposition and the visitor type in the case of functional decomposition. Extensions in the dual dimension require a bit more overhead than extensions in the primary dimension. In particular, the merge of independent extensions in the dual dimension requires a deep mixin composition as compared to a shallow mixin composition for a merge in the primary dimension.

This principle applies to several variants of operations: simple operations that access the tree only as the receiver of operation methods, tree transformers that return trees as results, and binary methods that take trees as additional arguments.

All implementation schemes discussed in this paper are sufficiently simple to be directly usable by programmers without special support for program generation. We conclude that they constitute a satisfactory solution to the expression problem in its full generality.

The examples in this paper also demonstrate that SCALA's abstract type members, mixin composition and explicitly typed self references provide a good basis for type-safe extensions of software systems. Other approaches to this problem have also been investigated; in particular family polymorphism [9] based on virtual classes [14] or delegation layers [18]. Compared with these approaches, SCALA's constructs expose the underlying mechanisms to a higher degree. On the other hand, they have a clearer type-theoretic foundation, and their type soundness has been established in the *vObj* core calculus.

Acknowledgments Philippe Altherr, Vincent Cremet, Burak Emir, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, and Erik Stenman have contributed to the SCALA design and implementation, which was partially supported by grants from the Swiss National Fund under project NFS 21-61825, the Swiss National Competence Center for Research MICS, Microsoft Research, and the Hasler Foundation. We also thank Philip Wadler, Shriram Krishnamurti, and Kim Bruce for useful discussions on the expression problem.

References

[1] G. Bracha. Personal communication, July 2002.

[2] G. Bracha and W. Cook. Mixin-based inheritance. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[3] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of OOPSLA '98*, October 1998.

[4] K. B. Bruce. Some challenging typing issues in object-oriented languages. *Electronic Notes in Theoretical Computer Science*, 82(8), 2003.

[5] K. B. Bruce, A. Fiech, and L. Petersen. Subtyping is not a good "Match" for object-oriented languages. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 104–127, 1997.

[6] K. B. Bruce, A. Schuett, and R. van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 27–51, 1995.

[7] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel. Towards a taxonomy of software change. To appear in *Journal of Software Maintenance and Evolution: Research and Practice (Special Issue on USE)*, 2004.

[8] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 130–145. ACM Press, October 2000.

[9] E. Ernst. Family polymorphism. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 303–326, Budapest, Hungary, 2001.

[10] E. Ernst. Higher-order hierarchies. In *Proceedings of the European Conference on Object-Oriented Programming*, LNCS 2743, pages 303–329, Heidelberg, Germany, July 2003. Springer-Verlag.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[12] C. Grothoff. Walkabout revisited: The Runabout. In *Proceedings of the 17th European Conference on Object-Oriented Programming*, Darmstadt, Germany, June 2003.

[13] S. Krishnamurthi, M. Felleisen, and D. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *European Conference on Object-Oriented Programming*, pages 91–113, 1998.

[14] O. L. Madsen and B. Møller-Pedersen. Virtual Classes: A powerful mechanism for object-oriented programming. In *Proceedings OOPSLA'89*, pages 397–406, October 1989.

[15] T. Millstein, C. Bleckner, and C. Chambers. Modular typechecking for hierarchically extensible datatypes and functions. In *Proceedings of the International Conference on Functional Programming*, Pittsburg, PA, October 2002.

[16] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. Scala distribution. École Polytechnique Fédérale de Lausanne, Switzerland, January 2004. `http://scala.epfl.ch`

[17] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In *Proceedings of the European Conference on Object-Oriented Programming*, Darmstadt, Germany, July 2003.

[18] K. Ostermann. Dynamically composable collaborations with delegation layers. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, Malaga, Spain, 2002.

[19] J. Palsberg and C. B. Jay. The essence of the visitor pattern. Technical Report 5, University of Technology, Sydney, 1997.

[20] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *Proceedings of the 17th European Conference on Object-Oriented Programming*, Darmstadt, Germany, June 2003.

[21] C. Szyperski. Independently extensible systems – software engineering potential and challenges. In *Proceedings of the 19th Australian Computer Science Conference*, Melbourne, Australia, 1996.

[22] M. Torgersen. Virtual types are statically safe. In *5th Workshop on Foundations of Object-Oriented Languages*, San Diego, CA, USA, January 1998.

[23] M. Torgersen. The expression problem revisited — Four new solutions using generics. In *Proceedings of the 18th European Conference on Object-Oriented Programming*, Oslo, Norway, June 2004.

[24] M. Torgersen, C. P. Hansen, E. Ernst, P. vod der Ahé, G. Bracha, and N. Gafter. Adding wildcards to the Java programming language. In *Proceedings SAC 2004*, Nicosia, Cyprus, March 2004.

[25] P. Wadler and et al. The expression problem. Discussion on the Java-Genericity mailing list, December 1998.

[26] M. Zenger and M. Odersky. Extensible algebraic datatypes with defaults. In *Proceedings of the International Conference on Functional Programming*, Firenze, Italy, September 2001.

[27] M. Zenger and M. Odersky. Implementing extensible compilers. In *ECOOP Workshop on Multiparadigm Programming with Object-Oriented Languages*, Budapest, Hungary, June 2001.