Aarhus Universitet
Datalogisk Institut
Åbogade 34
8200 Århus N

May 31, 2005

Types in Object-Oriented Languages
The Expression Problem in Scala

{**esbenn, mcalster, sm, argo**}**@daimi.au.dk**

Esben Toftdahl Nielsen	20001722
Kim Alster Larsen	20011709
Søren Markert	20010467
Kristian Ellebæk Kjær	20001960

This report contains 21 numbered pages.

Source code is available at `http://www.daimi.au.dk/~esbenn/tool`

All group members have contributed equally to the project.

Contents

1 Introduction

In this report we analyze and compare a number of solution proposals to "The Expression Problem". We have chosen Scala [10] as a base language for analyzing and comparing the solution proposals.

Our goal is to evaluate 4 different partial solutions to the expression problem in terms of extensibility, independent extensibility, binary methods, and ease of application. Extensibility refers to how easy it is to extend a language in both dimensions. Independent extensibility is whether it is possible to combine independently developed extensions, and how easy it is. Binary methods refers to the possibility of adding binary methods to the language. Finally, ease of application is a, subjective, evaluation of how easy the proposal is to use in practice.

To equal the playing field, all proposals have been implemented in Scala. Scala was chosen because of its rich set of language constructs. Scala supports both virtual types and parametric types, which are used in different proposals. Besides, Scala offers mixin-composition, which is necessary for the ability to combine independent extensions.

The first two of the solutions are from [9], and focus on independent extensibility when using Scala to implement solutions to the expression problem. The first is an object-oriented decomposition (OODC), and the second is a functional decomposition (FDC) using a visitor pattern [7]. The third solution is from [12], which covers 4 different proposals. We have only looked at the last of these proposals, which is a hybrid between an object-oriented and a functional decomposition (Hybrid). The last solution is from [3] and based on higher-order hierarchies (HoH) [4].

In the following section we give a brief overview of the problem in focus, the language in use and describe essential concepts used in the report. In Section 3 we analyze each proposal separately and subsequently compare these in Section 4. Section 5 describes related work and finally Section 6 concludes.

2 Knowledge Foundation

In this section we present a formulation of the expression problem, describe the implementation language, which we evaluate the solutions in, and describe some essential concepts.

2.1 The expression problem

The expression problem, originally formulated by Wadler [14] and since reformulated by numerous participants in the solution battle, is inherently controvert. We take no particular stand in how exactly to describe the problem, rather we refer to [3], [9] and [12] as a basis for this paper.

However we want to explicate the following simple formulation by Torgersen: *"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.". We stress "without runtime type errors" as an important statement because the expression problem really becomes interesting (read challenging) when exposed to a type system that ensures safe execution of the code.

As the above formulation indicates, the expression problem is about two dimensional extensibility. If a data type is modeled in an object-oriented fashion it is easy to add additional data types by simply by adding new classes, but adding a new operation over the data types requires adding new methods to each existing class (and thus altering existing source code). If the data type is modeled in a functional fashion (**SML** style) it is easy to add a new operation over the data types, but adding new data types requires rewriting each existing operator over these types. Naturally, a valid solution to the expression problem must allow both data extensions and operation extensions at the same time without modifying existing code.

2.2 Scala

Scala is an object-oriented (and to some extent also functional) statically typed programming language. The reason for choosing Scala as a basis for analyzing the solutions to the expression problem is that it contains most of the concepts that have been used in the solutions that we look at. At the same time it maintains a type system that ensures that entities are used in a type-safe fashion at runtime. Scala's type system supports parameterized classes, virtual types, explicitly typed self references, mixin compositions and more, all are part of the different proposals that we analyze.

2.3 Concepts

Here we will explain some basic concepts and terminology related to the expression problem and the discussion of it.

Parameterized classes Some of the solutions we discuss depend on different kinds of genericity. Parameterized classes is one such kind and it allows classes to be parameterized by a data type as shown in the following example.

```
VehicleList[E <: Vehicle] = {
  get(): E;
  put(e: E);
}
```

Here `E` is the type parameter. A parameter can be *bounded* by a type, in this case `Vehicle`, which means that `E` has to be a subtype of `Vehicle`. Alternatively `E` could be *unbounded* and thus any type would be accepted as parameter and in this case the `<:Vehicle` would have been omitted from the above example. In Java an unbounded type parameter is implicitly bound by `Object`. Essentially

parameterized classes are function that take types as arguments and return a class, and one of the solutions we shall see later is based on this language mechanism.

Recursive classes can be written in a type safe way by using *F-bounded polymorphism*. This allows a type parameter to appear in its own bound.

Virtual types Another way of handling genericity is virtual types. Virtual types were first introduced in BETA [8]. The general idea is similar to that of parameterized classes, but instead of giving the types as parameters, a class contains a type variable. The mechanism is best explained through an example:

```
VehicleList = {
  type E <: Vehicle;
  get(): E;
  put(e: E);
}
```

Here `E` is a virtual type variable, and is defined to be a subclass of `Vehicle`. The `<:` means that the type is virtual and that it can be further bound. The easiest way to think of virtual types is as aliases of classes. I.e, `E` above is an alias for the class Vehicle, which all the elements of the list have to be a subclass of. The `List` can be further bound:

```
BikeList = VehicleList {
  E <: Bike;
  ...
}
```

This is not statically type safe, the following example clarifies:

```
ListPutter = {
  putStuff(l: VehicleList, stuff: Vehicle){
    l.put(stuff);
  }
}

BikeList bl = new BikeList();
ListPutter lp = new ListPutter();

lp(bl,new Bike()); //Ok

lp(bl, new Car(); //Wrong
```

If the arguments to `putStuff` is of type `BikeList` and `Bike` the method `putStuff` works fine, but it is not possible to check this statically. If something other than a `Bike`, e.g a `Car` is given as the second argument, a runtime error will occur. BETA solves this by generating runtime checks. Torgersen [13] argues that if the `ListPutter` problem gives rise to a type-check error instead of a runtime check, virtual types are statically type safe. He also shows that the `ListPutter` example can be rewritten to avoid the problem.

As mentioned previously Scala offers virtual types. In Scala, however, the type variable must be finally bound, in order to instantiate a class. This is due to the implementation of virtual types in Scala, depending on abstract classes in Java.

Shallow and deep mixin composition When doing mixin composition in Scala we refer to a composition where we only combine traits without the need to specifically state the relations of the classes contained in the traits as *shallow mixin composition*. When it is necessary to combine each individual class and trait explicitly inside the new trait (or class) we refer to it as *deep mixin composition*. We have borrowed both of these terms from [9]

3 Analysis

In this section we analyze four different proposed solutions to the expression problem. All proposals start with a base language consisting of numerals, and extends them with additional functionality. We have implemented all proposals and performed experiments on them. Amongst those, we have tried a few things not discussed in the articles in which they were introduced.

3.1 Object-oriented and functional decomposition

Odersky & Zenger [9] propose two partial solutions to the expression problem in the Scala language using virtual types. One proposal is based on object-oriented decomposition while the other is based on functional decomposition. The solutions focus primarily on the support of virtual types, and one uses the visitor-pattern for functional decomposition.

3.1.1 Object-Oriented decomposition

This approach is similar to a standard decomposition of expressions in object-oriented languages, using the *Interpreter* design pattern [7], in that it introduces an abstract `Exp` type, with concrete expressions as subclasses of `Exp`. Operators are defined as methods on each class. To support independent extensions without changes to existing code, virtual types and Scala's mixin capabilities are used.[1]

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

Figure 1: The Base language of OODC

[1] Implementation is in file `ObjectOrientedDecomposition.scala`

The approach is to use traits with a virtual type `Exp`. Each trait contains the classes of the datatypes of a given language, each extending a basic `Exp` and containing operators on these datatypes, see Figure 1. To extend the language, its traits must be extended and then in the extending trait the virtual type must be further bound to the extended version of `Exp`.

An extension of the language with a new datatype amounts to extending the trait, and adding a new class for the new datatype.

Extending with a new operator amounts to extending the base trait and inside that extending the `Exp` class and each class in the trait with the new operator. Although this is not quite as easy as extending with a new datatype, it is as easy as would be expected when using object-oriented decomposition. Since all operators are defined in the classes of the individual datatypes, it has to be added to each.

Traits are abstract in the sense that they cannot be instantiated. To use a language a new class extending the trait of the language has to be created. In this class the virtual expression type is finally bound to `Exp`, it will then correspond to the last version of `Exp` in all operators and classes.

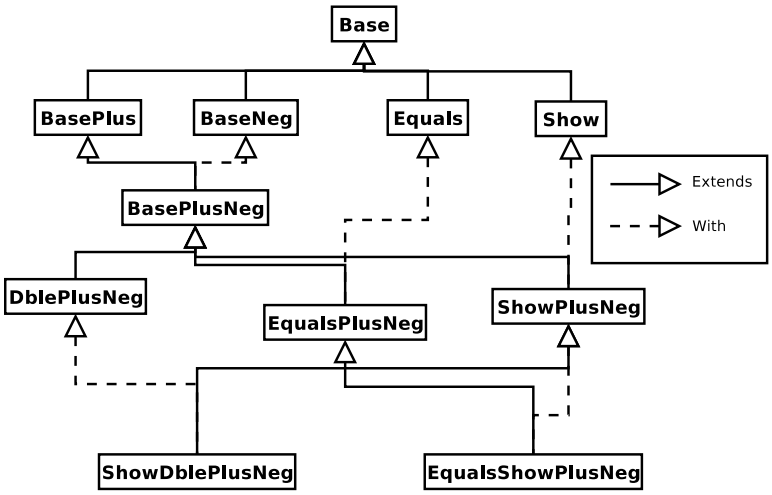

Figure 2: Extensions to the OODC based language.

Adding new datatypes and new operators is relatively easy, but not much more so, than it would be in other object-oriented languages. The interesting part of this approach is the combination of independent extensions. The authors demonstrate this several times. First by extending the basic language, consisting only of literals, with `Plus` and `Neg` data types, and then combining them to a new language. They also show that it is possible to mix datatype extensions and operator extensions by adding a `show` operator to the language, and combining it with the data extensions. See Figure 2 for an overview.

The approach is really the standard object-oriented decomposition translated to Scala. Because of Scala's expressiveness and use of mixins you get the possibility

of combining independent extensions, but not always in a trivial way. When combining operator extensions it is necessary to do *deep mixin composition*, meaning that it is not enough to combine traits, you also have to combine each class in the trait.

When adding operators that transform the tree, it is also necessary to add factory methods that return `exp`, but this is true for all statically typed languages, since it would otherwise be impossible to substitute e.g. a `Plus` class with an `Add` class, when they have no subclass relation between them.

Binary methods are a problem in themselves, and in the object-oriented approach it is shown that this is relatively easy to add because of virtual types. For example, an equals method can easily be made to only accept objects of the same type by finally binding its virtual type variable.

3.1.2 Functional decomposition

The functional decomposition uses the visitor-pattern to simulate functional decomposition of the given language, see Figure 4. This is the standard approach when using object-oriented languages[2].

Extending the language with a new operator can be done with very little work, simply by writing a new visitor, which performs the new function on the visited expressions. Extending with a new datatype, however, is at bit more laborious. The `Visitor` trait has to be extended with a function handling the case with the new datatype.

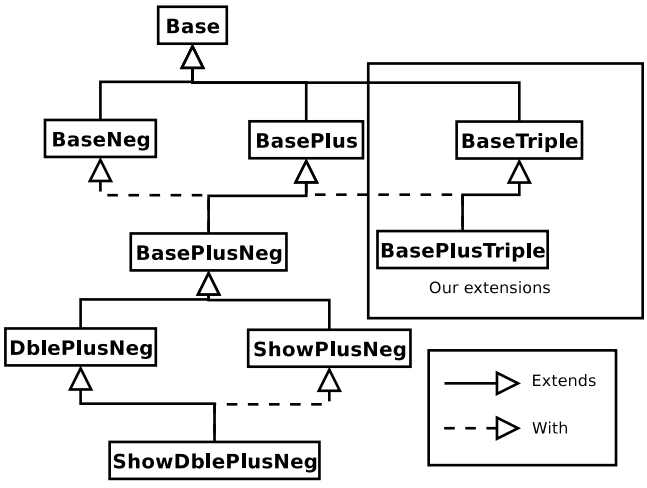

Figure 3: Extensions to the FDC language

As would be expected, in the functional decomposition combining independent function extensions is easy, but combining data extensions require *deep mixin*

[2]Implementation is in file `FunctionalDecomposition.scala`

```
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;
    }
  }
}
```

Figure 4: FDC base language

composition to combine the visitors from both extensions. Essentially, to combine independent operators, you only have to mix their respective traits:

```
trait ShowDblePlusNeg extends DblePlusNeg with ShowPlusneg;
```

Unlike the object-oriented approach, it is not shown that it is easy to combine independent data extensions with independent function extensions in the functional decomposition. Instead independent operations to the same language are added and those are combined (Figure 3). To show that it is possible, we have added an operator `Triple` to the base language, and combined it with the language containing a `Plus` datatype. Our addition is marked by a square in Figure 3. This combination was shown to be quite easy (see Figure 5), although [9] does not even mention whether it is possible or not.

```
trait BasePlusTriple extends BaseTriple with BasePlus {
  type visitor <: Visitor;

  class Triple: visitor extends super.Triple with super[BasePlus].Visitor{
    def visitPlus(l: Exp, r: Exp) =  {
      result = new Plus(apply(l),apply(r));
    }
  }
}
```

Figure 5: combining data and operator extensions in FDC

In [9] the task of adding binary operators to the functional decomposition is left as an exercise to the reader. However, it is not immediately clear, at least not to us,

how to accomplish this. It seems that, although a solution is most likely possible, it is not simple.

3.1.3 Commonnalities

[9] is not concerned with much else than the expression problem itself. Specifically it does not address important aspects of partial solutions like reuse of creation code and level of extensibility (see Section 4.1).

Another important aspect that is overlooked is how to structure the extensions to facilitate combination of independent extensions of more than one level. Because of Scala's requirement that a mixed-in class' direct superclass must be a superclass of the class it is mixed with, Scala imposes some limitations as to how extensions can be further extended. This can be overcome though, by making sure that all extensions have the `Base` language as a direct superclass, and we will show how to do this in section 4.2. So although [9] does not mention this, the proposals can easily be refactored to allow it. It will however, give rise to slightly more complex code, since it has to be specified that new languages extend the expressions from the mixed-in class(es) and not from the superclass.

3.2 A hybrid solution

In [12] Torgersen presents four new solutions to the expression problem using Java generics. We will only treat the last of the four solutions proposed in the paper, the hybrid, since it is the most advanced solution of the four and serves as a good source of comparison to the other solutions we consider[3].

The hybrid solution in Figure 6[4] combines the data-centered approach with the operation-centered approach. The solution uses the Visitor pattern for keeping operation extensibility and an abstract parameterized class `Node` to keep the data type extensibility. Torgersen's objective is to create a framework that is extensible in both dimensions without needing to modify existing code and keeping all three levels of extensibility, see Figure 8.

The solution uses parameterized classes to be able to leave the framework open for later extension. Take the following example:

```
abstract class Node[V <: Visitor] extends Expression { ... }
```

The `Node` class is a common super class of all language datatypes, it is parameterized with a type `V` whose type is open, except that it has to be a subclass of the trait `Visitor`. This will let future extensions create their own `Visitor` implementations and thereby create subclasses of `Node` using this particular `visitor`.

[3] Implementation is in file `Hybrid.scala`

[4] This is the Scala version of Torgersens original Java-based implementation. Note that this example will not work as intended in the current implementation of Scala. This is because `isInstanceOf` is not properly implemented yet. In the example source code of our tests we overcame this with a cast and a catch as can be seen in `Hybrid.scala`.

```
trait Base {
  trait Expression {
    def handle(v: Visitor): unit;
  }

  trait Visitor {
    def apply(l: Expression): unit;
    def default(l: Expression): unit;
  }

  abstract class Node[V <: Visitor] extends Expression {
    def handle(v: Visitor): unit =  {
      if (v.isInstanceOf[V]) {
        accept(v.asInstanceOf[V]);
      } else {
        v.default(this);
      }
    }
    def accept(v: V): unit;
  }

  abstract class Op[E <: Expression] extends Visitor {
    def apply(exp: Expression): unit = {
      if (exp.isInstanceOf[E]) {
        call(exp.asInstanceOf[E]);
      } else {
        exp.handle(this);
      }
    }
    def call(e: E): unit;
    def default(e: Expression): unit = {
      throw new IllegalArgumentException("Expression problem occurred!");
    }
  }
}
```

Figure 6: The Base language of Hybrid

In our implementation of Torgersen's framework in the context of Scala we have introduced a grouping of language hierarchies, something that is possible in Scala by use of *trait*s to create mixin compositions. As mentioned above this is the same way Odersky & Zenger have structured their proposals in order to be able to create independent extensions that can later be combined. Later we will explain how exactly to combine independent extension consisting of several levels of extensions, something not clarified in any of the solution proposals.

As illustrated in Figure 6 every language extension is grouped into a *trait* that can be extended further. Then later the framework is used like this:

```
trait LanguageWithEval extends Base {
   // new language constructs go here
}
```

Apart from this grouping mechanism we have implemented the hybrid solution as proposed and in the following we will analyze what actually makes it work. The

hybrid solution uses Java-style parameterized classes to make a generic framework for creating expressions belonging to a certain language.

The `Node` class in Figure 6 is parameterized with a type parameter `V` bounded by `Visitor`. The reason for this is that then it is possible for later extensions of `Node` to handle extensions of the `Visitor` class. As can be seen from Figure 7 class

```
trait Ale extends Base {
  trait PrintExp with Expression {
    def print(print: Print): unit;
  }

  class Print() extends Op[PrintExp] {
    override def call(e: PrintExp): unit = {
      e.print(this);
    }
  }

  trait AleVisitor with Visitor {
    def visitLit(lit: Lit): unit;
    def visitAdd(add: Add): unit;
  }

  class Lit(v: int) extends Node[AleVisitor] with PrintExp {
    var value: int = v;
    def print(print: Print): unit = {
      System.out.print(v);
    }
    def accept(v: AleVisitor): unit = {
      v.visitLit(this);
    }
  }

  class Add(l: Expression, r: Expression ) extends
    Node[AleVisitor] with PrintExp { ... }
}
```

Figure 7: The ALE language

`Lit` is an extension of `Node` with the parameterized type bound to `AleVisitor` - the type system now knows that it is safe to call the method `visitLit(this)` on an `AleVisitor` instance.

For the hybrid solution to be able to handle all datatypes from previous versions of the language it makes a runtime type-check of the argument expression. If it is not of a known type (i.e. E) then we invoke the method `handle(this)` to handle older versions of abstract syntax trees. Class `Op` in Figure 6 implements an `apply(...)` method for applying this operation on an expression. If the argument expression is of an older version (it does not know how to handle it) the operation surrenders the responsibility to the expression with itself (`this`) as call-back (double dispatch), which then invokes `accept(v)`. This will result in the correct visit method (this `accept` method forwards its calls to the `visit` method of the visitor with itself as an argument) to be invoked. If the visitor instance passed to `accept`

is not recognized (e.g. it is not subtype of `V`) the method `default` is called to stop the dispatch process (in hybrid it throws an exception). Example: When two independent extensions A and B of the same language where A has a functional extension which is applied to datatype from language B, it will not be possible for the datatype to handle the visitor, and the exception is thrown. In Scala this limitation can be overcome by combining the two extensions hence making the types match. In Section 4.2 we will provide an example of such a combination in Scala.

Torgersen takes a slightly different approach to the problem than the other proposals in that he allows casts for the benefit of other things. In the Hybrid framework casts have been allowed in order to be able to achieve object-level extensibility, which from a reuse perspective could be very important. Furthermore if we look at the hybrid solution in Figure 6 it circumvents the type-system in a safe manner, in that it uses `instanceof` to ensure that it won't result in a runtime error. If the framework cannot ensure safe runtime-execution it gives up, resulting in an exception from the framework. We don't consider this exception a runtime type error, but a flaw in the use of the hybrid framework. During our implementation and experimentation with the hybrid solution in Scala we have come across this exception a couple of times as a result of incomplete combinations of languages as the only cause.

3.3 Higher-order Hierarchies

During our selection of solutions to the expression problems a recent solution by Ernst [3] puzzled us because of its immediate simplicity and on the same time being statically typed. The problem is implemented in *gbeta* [6] a generalization of the BETA, but we found it relevant to look at his solution from the context of Scala and therefor we analyze our findings here[5].

One significant difference between our solution and the one presented in [3] is that all inheritance combinations of member classes have to be declared explicit. This means that we have to repeat the relation between datatypes, like for example:

```
class Lit(value: int) extends Exp with super.Lit(value)
```

It can also be seen from the above example that the relation between `Lit` and `Exp` has to be reestablished explicitly. In *gbeta* there would actually be a relation between these classes other than that they just happen to be two classes with the same name in different scopes. This means that it is a cumbersome task to slide a new class in between two already defined classes in Scala, whereas in *gbeta* it would require nothing more that creating a new group and sliding it in between - the member classes of the group does not have to altered, which isn't the case in Scala.

Scala does not have higher-order hierarchies and family polymorphism [5], which makes the semantics of our Scala implementation somewhat different hence

[5] Implementation is in file `HigherOrderHierarchies.scala`

it difficult to make a valid comparison of the HoH solution to the other three solution proposals.

3.4 Type groups

In all the implementations of the proposals we have made we have used *type groups* to group together families of mutually dependent types e.g. types belonging to a specific language in the expression problem. Type groups are not first class entities in Scala, but the *trait* construct can be used to make the same grouping (mixin composition). Take the following example from the implementation of the hybrid approach:

```
trait Base {
   trait Exp {
     def handle(v: Visitor): unit;
   }
   ...
}
```

`Base` is a group of classes/types that is open for later refinement, but an important difference between the grouping mechanism proposed as an extension to *LOOJ* and Java in [1] and the way we use it in Scala is that the relation between members of type groups has to be declared explicitly as described in the previous section. Consider the following example.

```
trait Base {
  class Exp {
    def foo: unit = {Console.println("foo")};
  }
}

trait Full extends Base {
  class Exp {
    def bar: unit = {Console.println("bar")};
  }
}
```

Scala's type system will not complain at all if exposed to the above example, furthermore it will **not** relate the types `Base.Exp` and `Full.Exp`, because members of groups `Base` and `Full` are completely independent of each other when not stated otherwise. What we really wanted was `Full.Exp` to be a refinement of `Base.Exp`, but this can be done by telling this to Scala's type system explicitly:

```
trait Base {
  class Exp {
    def foo: unit = {Console.println("foo")};
  }
}

trait Full extends Base {
  class Exp extends super.Exp {
    def bar: unit = {Console.println("bar")};
  }
}
```

We think that the approach of having to explicitly state dependencies as in Scala is a both good and bad, on one side it prevents the programmer from accidentally relating classes from different groups that should not have been related, on the other hand it is cumbersome task to make changes to previous groups (as discussed in Section 3.3), because this will invalidate the whole group structure.

4 Comparison

In this section we will compare the different approaches to solving "The Expression Problem". The main focus will be on how the solutions differentiate from each other, and what kind of trade offs have been made. This section is divided into smaller topics and for each topic, there will be an analysis of the relevant parts of the four solutions. When applicable we will give references or comments on our experiences in implementing the different solutions.

Most importantly we will compare the four analyzed solutions from four different perspectives. These are *level of extensibility*, *indendent extensibility*, *binary methods* and *ease of application*.

4.1 Level of extensibility

Of the four approaches [12], and thereby the Hybrid solution, is the only one which considers the perspective of extensibility at different levels the most. From a code reuse perspective this dimension is very important because if code is not extensible at an appropriate level it might render itself useless to a potential user. As previously described Hybrid achieves object-level extensibility, which is necessary in the context of object persistence.

Figure 8: Levels of extensibility

Odersky and Zenger do not concern themselves with levels of extensibility. Is is clear that their code is source level extensible, but they do not discuss the other levels. Currently it is not possible to do mixin composition in Scala without having the source code of the mixed-in class, but this is only a limitation to the current implementation. When this is addressed appropriately all four of the addressed solutions become binary level extensible.

4.2 Independent extensibility

The criterion of independent extensibility is added to the definition of the expression problem in [9]. In this section we will look at how well the four proposed

solutions combine independent extensions. Figure 9 shows the simplest way a (base) language can be extended with two independent additions – they are both unaware of each other. At the bottom they are combined into one language that has the functionality of both extensions.

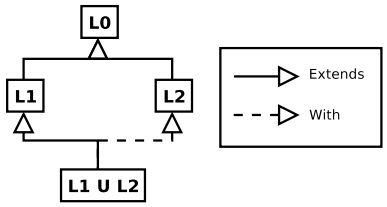

Figure 9: Combining independent extensions

Object-Oriented decomposition The OODC solution is shown to be able to combine independent extensions, both when the extensions combined are data extension and when they are operation extensions. They also show that a data extension can be combined with an operator extension.

However, they do not address the problem of combining independent linear extensions of more than one level. Although we have not tried this, it is quite easy to re-factor the code to allow combinations of linear extensions of more than one level, but it will require specifying exactly which language is extended in each extension, and thus gives slightly more complicated code.

Functional decomposition The FDC solution is shown also to have the same ability to combine independent extensions, but it has not been shown that it can combine an operation extension with a data extension. We have tried this, and showed it to be quite simple (see Figure 5). To combine a data extension with an operator extension, it is only necessary to extend each operator with handling of the new datatype.

The problem of combining independent extensions of more than one level is not handled here either, but as with OODC it is also possible with slight modification of the code.

Hybrid solution In [9] Odersky & Zenger claim that Torgersen omits considering how his solution would be able to combine independent extensions. It is true for as far as Java implementation goes that independent extensibility would be complicated maybe even impossible, but when the hybrid solution is implemented in Scala, it achieves full capability to combine independent extensions[6]. See Figure 10.

[6]Implementation is in file `HybridIndependent.scala`

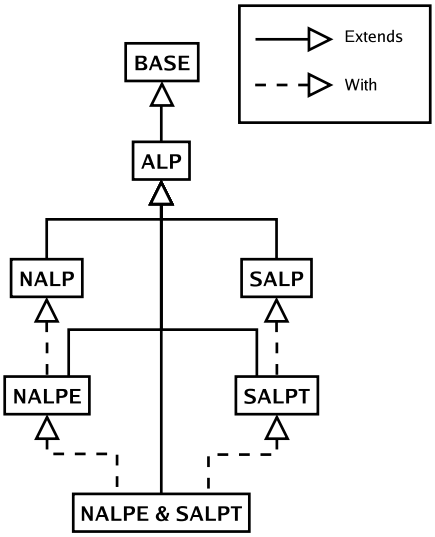

Figure 10: The “SNALPE” hierarchy

The base language has been extended by two levels with operation and data extensions, and finally those have been combined into one language containing all data variants and operations over them. That is the language at the bottom of the graph. Furthermore member classes `Node` and `Visitor` have to be mixed in as well, previously described as *deep mixin composition*. This mean that the member classes need to have the same structure as the language traits.

Higher-order Hierarchy based The HoH solution is also shown to have full capability to combine independent extensions. We have not made extensions to a language based on the HoH solution to the same extent as to the hybrid solution, but there is no indication that this solution should be less capable of doing so.

A note on Scala as implementation language As mentioned above, the step from Java to Scala as implementation language for the hybrid solution enabled it to combine independent extensions. As can be seen from Figure 10 each extension *extends* the base language and only inherits the added functionality in the extension languages. This has to be this way, because mixin in Scala requires that the direct super class of the mixed-in class is also a super class of the extended class.

4.3 Binary methods

The four solutions proposed all involve simple methods on the data objects i.e. methods taking no arguments or methods taking simple arguments such as a pretty-printers indentation level. In this subsection we will look at each of the solutions’ abilities to handle binary methods i.e. methods on the data objects that take the

same kind of data objects as an argument. We will use the method `eql`, comparing two expressions for equality, as a reference example.

Object-Oriented decomposition The OODC solution was extended with an `eql` method and then both data and operation extensions were added. The language containing the binary method is not extended independently and the extensions combined, however, so it is not known to be possible. It is, however, mixed with `BasePlusNeg` trait and the resulting trait is mixed with the `ShowPlusNeg` trait, so at least it is shown that it can be combined with a language containing datatype extensions and with a language containing operator extensions.

Functional decomposition The above mentioned implementation of `eql` was not made for the FDC solution. We attempted to use a similar approach, but it turned out that we needed the type variable that the OODC solution has. This was not desirable, since it would turn the FDC into a hybrid object-oriented and functional decomposition solution, and the solution would no longer be the same. So far it has not been shown to be possible, although it most likely is. Attempts of using a double dispatch approach have also failed, because of the need to define two different kinds of visitors.

Hybrid solution We implemented `eql` in the hybrid solution of using the same basic idea from OODC in [9] by adding a running update of the method-owning object to the visitor. It turned that this approach would work just fine when we tested it, but it is a non-trivial implementation[7].

Higher-order Hierarchy based We did not look into how well the solution of [3] handles binary methods, but it would be an interesting study.

It would also be interesting to look at how binary methods could be generalized to methods taking any number of arguments of any type would be handled in each of these solutions.

4.4 Ease of application

The amount of work involved in using each of these four solutions and the complexity of applying the solution are as important as the previous perspectives.

Object-Oriented and Functional decomposition The OODC and FDC solutions are quite simple to extend languages with. In OODC a data extension amounts to simply adding the new expression classes implementing all existing methods, and an operation extension requires a binding of the type variable and extending

[7] Implementation is in file `HybridBinaryOperations.scala`

the data type classes with the new method. In FDC a data extension requires extending the base expression implementing visitor functionality, but much less so than in the hybrid solution. An operation extension requires extending the visitor to a new kind, defining methods to handle each existing data type. When it comes to implementing binary methods, the OODC solution is reasonably simple to work with, but the FDC solution caused quite a lot of trouble. All in all these two solutions are simpler than the hybrid solution and takes less work.

Hybrid solution Working with the hybrid solution is not trivial. It is complex in its mechanics and extending languages based on it requires of the programmer to either have a good understanding of how the solution works or to follow the examples in [12] closely. In order to create a base language, the expression interface must be extended, the visitor interface must be extended, data types must extend `Node` with visitor handling functionality and finally operations must extend `Op`. Then in order to extend the language with an operation, the expression interface must be extended to another level and `Op` must be extended implementing visitor functionality. A data extension has to extend `Node` with visitor functionality and extend the visitor interface to another level. Implementing a binary operation in the hybrid solution is even more complex. It requires the visitor to update its own reference to one if its sub-expressions each time it is passed on to the next sub-expression of the argument expression. All in all a quite complicated and cumbersome approach to work with.

Higher-order Hierarchy based In the HoH solution extending a language with either an operation or a data type amounts to simply extending the class of the language and then either adding a new component class in case of a data extension or extending all the component classes with a new method each. It cannot possibly be any simpler than this. Binary methods were not implemented in this solution, so we have no indication of the simplicity of and work involved in such additions.

Also, Scala does not implement family polymorphism and higher-order hierarchies which makes it difficult to make a valid comparison of the problem implemented in *gbeta* and the one we have implemented in Scala.

5 Related work

Aspect Oriented Programming (AOP) The goal of Aspect Oriented Programming is typically considered improving separation of concerns and most AOP approaches provide possibility for this by means of instruments to modify existing code without actually editing the code. As pointed out in [4] and [12] a solution to the expression problem can be made readily by use of AOP because of this possibility to inject new code into old code. Unfortunately this approach is only source-level extensible because injecting new code into old code requires recompilation and hence is not re-usable at binary level. We still think this approach is

interesting because it introduces a new dimension to the expression problem that the above considered proposals have not favored; namely separation of concerns. In contrast, and also underlined in Section 4.4, they very much **tangle** concerns in the two dimensions that they try to solve the problem in. With AOP it would be possible to completely separate extending new datatypes from extending with new operations over them, which from a reuse perspective might be important.

This illustrates the core dilemma of the expression problem: It is all about compromise. As described above it is possible to solve the expression problem using AOP with static type safety and the ability to extend in both dimensions, but with the compromise of only having source-level extensibility.

Structural Virtual Types Structural Virtual Types are presented in [11] as a merger of parameterized classes and virtual types, which provide the same expressiveness as both F-bound polymorphism and virtual types (i.e. three dimensions of subtyping) and are statically typed. It is an open question whether structural virtual types could give rise to an improved solution to the expression problem.

6 Conclusion

Four solutions to the expression problem have been treated from the point of view of the Scala programming language. Scala's abilities of expression has helped us implement a version of each solution and extend the solutions with further additions that were not considered in their original settings. Scala has proven to be a good base for solving the expression problem. Working with it has made it easy to implement and experiment with the problem at hand and the solutions we have analyzed.

We have looked at three categories of solutions, namely those based on *virtual types*, *parameterized types* and *higher order hierarchies*.

We have shown that the Hybrid solution originally written in Java, that was unable to perform combinations of independent extensions, is able to do so when implemented in Scala. Thus the chosen implementation language and its expressiveness is a major factor in deciding how well programmers will be able to perform two dimensional extensions in their applications.

Regarding level of extensibility, all of the approaches support source level extensibility. This can be seen as the minimum requirement for a solution to the expression problem. They also, in theory, support binary level extensibility, when Scala is fully implemented. The Hybrid solution is the only one supporting object level extensibility, but without statical type safety all though it does so in a safe manner as explained earlier.

All of the solutions are independently extensible with both new operators and new datatypes, and they can all combine independent extensions of multiple levels[8]. However, all of the solutions require *deep mixin composition* to combine at

[8] OODC and FDC would have to be slightly refactored to allow this.

least some extensions, except for HoH in *gbeta*, but only because *gbeta* provides direct language support for deep mixin composition.

We have looked at binary methods in three proposals. In OODC and Hybrid adding binary methods is possible, but non-trivial. Implementation-wise, binary methods in OODC is trivial, but the idea used in [9] does not appear simple. We tried adding binary methods to FDC, first using the same approach as in OODC, and later using a double-dispatch approach, but we could get none of them to work without changing existing code. Of course, this might just be because we could not get the right idea. As mentioned we managed to extend Hybrid with a binary operator `eql` by piggy-bagging the visitor down the expression tree, but it was rather complicated. We have not looked into binary methods in HoH.

OODC, FDC and HoH are all relatively easy to use and extend. The code is not too complicated, perhaps except for the visitor based FDC approach, which has a control flow that is not immediately clear, but this is true for most visitor based programs. The Hybrid solution requires a lot of complicated code which reduces readability drastically. Independent combinations add to the complexity of all solutions. None of them are straight forward to implement with support for this.

Of the four different solutions, Hybrid is the most general. It has extensibility on all levels, and it also considers reuse of client code. The price is that Hybrid has increased complexity and that it is not completely statically type safe. The complexity alone limits the use of the framework. If the increased flexibility is not imperative, it is probably easier to use another, more simple approach, such as OODC or FDC.

References

[1] Bruce K.: *Some Challeging Typing Issues in Obejct-Oriented Languages*, Elsevier Science B.V, 2003.

[2] Bruce K. *et al.*: *On Binary Methods*, Theory and Practice of Object systems 1 (3)

[3] Ernst E.: *The expression problem, Scandinavian style*, ECOOP'04.

[4] Ernst E.: *Higher-Order hierarchies*, Proceedings ECOOP'03.

[5] Ernst E.: *Family Polymorphism*, Proceedings ECOOP'01

[6] Ernst E.: *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. Ph.D. thesis. Department of Computer Science, University of Aarhus, Denmark, 1999.

[7] Gamma, E *et al.* (1995) Design Patterns: Elements of reusable Object-Oriented Software. Addison-Wesley

[8] Madsen *et al.*: *Object-Oriented Programming in the BETA Programming Language*, Addison-Wesley, 1993.

[9] Odersky *et al.*: *Independently Extensible Solutions to the Expression Problem*.

[10] Odersky *et al.*: *An Overview of the Scala Programming Language*.

[11] Thorup K. K., Torgersen M.: *Unifying Genericity - Combining the Benefit of Virtual Types and Parameterized Classes*.

[12] Torgersen M.: *The expression problem revisited - four new solutions using generics*.

[13] Torgersen M.: *Virtual Types are Statically Safe*.

[14] Wadler P.: *The Expression Problem*. Posted on the Java Genericity mailing list, 1998.