Modular Visitor Components

A Practical Solution to the Expression Families Problem

Bruno C. d. S. Oliveira

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
bruno@comlab.ox.ac.uk

Abstract. The *expression families problem* can be defined as the problem of achieving *reusability and composability* across the components involved in a *family* of related datatypes and corresponding operations over those datatypes. Like the traditional *expression problem*, adding new components (either variants or operations) should be possible while preserving *modular and static type-safety*. Moreover, different combinations of components should have different type identities and the subtyping relationships between components should be preserved. By generalizing previous work exploring the connection between type-theoretic encodings of datatypes and visitors, we propose two solutions for this problem in Scala using *modular visitor components*. These components can be grouped into *features* that can be easily composed in a *feature-oriented programming* style to obtain customized datatypes and operations.

1 Introduction

Component-oriented programming (COP) [1], a programming style in which software is assembled from independent components has, for a long time, been advocated as a solution to the so-called *software crisis* [2]. However, the truth is that to date the COP vision has not been fully realised, largely due to limitations of current programming languages. A particular problem is that most languages have a bias towards one kind of decomposition of software systems, which imposes a corresponding bias on the kinds of *extensibility* available [3, 4]: in same languages adding new datatype variants is easy, while in others adding new operations is easy. Providing software systems that support both kinds of extensibility at the same time has proved itself quite elusive to achieve in existing languages and leads to what Wadler calls the *expression problem* [5].

In this paper we will look at a variation of the expression problem (EP) that we call the *expression families problem* (EFP). The EFP can be defined as the problem of achieving *reusability and composability* across the components involved in a *family* of related expression datatypes and corresponding operations over those datatypes. Like with the traditional EP, adding new components (either variants or operations) should be possible while preserving *modular and static type-safety* (that is, no modification or duplication and no re-compilation and re-typechecking of existing code should be needed). Furthermore, it should

also be possible to combine independently developed extensions [6]. Additionally, a solution to the EFP should: allow different combinations of components to have *different type identities*; *preserve the subtyping relationships* between the different components (whether in the same family or a different one); and *provide a high degree of composability and decoupling of components*.

By generalizing previous work [7, 8] exploring the connection between type-theoretic encodings of datatypes [9, 10] and the VISITOR pattern [11], we propose two solutions for this problem in Scala[1] using *modular visitor components*. These components can be grouped into *features* that can be easily composed in a *feature-oriented programming* style. The solutions presented in this paper do not require any extensions to Scala and rely only on features that, although not yet widely available in mainstream OO languages, have been shown to be independently useful in the past. In particular, we make use of the following features: *higher-order type parameters* [12, 13], *traits and mixin composition* [14, 15], *self-types* [16] and *variance annotations* [17]. Of these, self-types are only required by one of the solutions and could potentially be completely eliminated using a technique devised by Torgersen [18]. The other three features are needed to address all the requirements of the EFP. However we should remark that variance annotations are only required to ensure that the subtyping relations between different datatypes are preserved, but otherwise they would not be necessary (in particular, they would not be needed to solve the traditional EP).

In Section 2 we motivate and formulate the expression families problem. The technical contributions follow:

- Section 3 shows how to adapt type-theoretic encodings of datatypes to support extensibility of variants as well as extensibility of operations.
- Section 4 shows a simple solution for the EFP inspired by Church encodings of datatypes. It is also shown how the subtyping relations between components of different families can be helpful for scalability and reuse.
- Section 5 shows another solution for the EFP inspired by Parigot encodings of datatypes. This solution is more expressive than the one in Section 4, but it is also slightly more complex to use.
- Section 6 shows how we can group the modular visitor components into features that can be easily combined by clients to obtain customized datatypes and operations.

A comparison between our work and solutions to the expression problem is presented in the Section 7. Conclusions are presented in Section 8.

2 The Expression Families Problem

In the expression families problem we are interested in modularizing and reusing the common parts of a family of expression datatypes and corresponding family of operations. For example, in some context, we may have a system composed of

[1] Source code available at: http://web.comlab.ox.ac.uk/people/Bruno.Oliveira/EFP.tgz

a datatype of expressions Exp_1 that supports numeric, addition and subtraction variants together with a corresponding evaluation function:

$$\begin{aligned}
&\textbf{data } Exp_1 = Num_1\ Int \mid Add_1\ Exp_1\ Exp_1 \mid Minus_1\ Exp_1\ Exp_1\\
&eval_1 :: Exp_1 \to Int\\
&eval_1\ (Num_1\ x) &&= x\\
&eval_1\ (Add_1\ e1\ e2) &&= eval_1\ e1 + eval_1\ e2\\
&eval_1\ (Minus_1\ e1\ e2) &&= eval_1\ e1 - eval_1\ e2
\end{aligned}$$

In a different context we may have a system composed of a datatype Exp_2 that also supports negation and provides both an evaluation operation and an operation that narrows Exp_2 expressions into Exp_1 expressions:

$$\begin{aligned}
&\textbf{data } Exp_2 = Num_2\ Int \mid Add_2\ Exp_2\ Exp_2 \mid Minus_2\ Exp_2\ Exp_2 \mid Neg_2\ Exp_2\\
&eval_2 :: Exp_2 \to Int\\
&eval_2\ (Num_2\ x) &&= x\\
&eval_2\ (Add_2\ e1\ e2) &&= eval_2\ e1 + eval_2\ e2\\
&eval_2\ (Minus_2\ e1\ e2) &&= eval_2\ e1 - eval_2\ e2\\
&eval_2\ (Neg_2\ e) &&= -\ (eval_2\ e)\\
&narrow_{21} :: Exp_2 \to Exp_1\\
&narrow_{21}\ (Num_2\ x) &&= Num_1\ x\\
&narrow_{21}\ (Add_2\ e1\ e2) &&= Add_1\ (narrow_{21}\ e1)\ (narrow_{21}\ e2)\\
&narrow_{21}\ (Minus_2\ e1\ e2) &&= Minus_1\ (narrow_{21}\ e1)\ (narrow_{21}\ e2)\\
&narrow_{21}\ (Neg_2\ e) &&= Minus_1\ (Num_1\ 0)\ (narrow_{21}\ e)
\end{aligned}$$

The two systems are clearly related and share a lot of code, but there is not any reuse of code (in a software engineering sense) between them. In current programming languages, achieving reusability between these two systems is not easy because datatypes and operations are evolving at the same time. This is, after all, the EP — we suggest [6] for a good introduction to the original EP for readers unfamiliar with it. However, there is something more about this example that is not normally emphasized in the context of the EP. The $narrow_{21}$ operation takes a value of Exp_2 and converts it to a value of Exp_1. Among other things, it is *statically* known that the result of $narrow_{21}$ will not contain any negation variant. Solutions for the EP are only required to allow extensibility, but there is no explicit requirement about the interaction between *distinct* types of expressions. In particular, this allows for solutions where there is only a single, global expression datatype [19–21]. However, with these approaches it is not possible to accurately express the type of $narrow_{21}$. Consequently these solutions fail to solve the EFP because they do not meet the following requirement:

Different kinds of expressions should have different type identities.

Another aspect about this example that is not normally emphasized in the context of the EP — although both Wadler [5] and Zenger and Odersky [20] do mention it — is that there are interesting subtyping relationships between some of the components *in different families*. In particular $Exp_1 <: Exp_2$ and $eval_2 <: eval_1$. More generally, the extension of a datatype becomes a *supertype* of the original datatype; while the extension of an operation becomes a *subtype*

of the original operation [22]. These relations are important for legacy and performance reasons since it means that, for example, a value of type Exp_1 can be *automatically* and *safely* coerced (at no run-time cost) into a value of type Exp_2, allowing some interoperability between new functionality and legacy code. This leads us to the following requirement for the EFP:

> *Subtyping relationships between components should be preserved.*

In our example we can identify a number of different features: on the one hand we have the set of operations $\{eval, narrow\}$ and, on the other hand, we have set of variants $\{Num, Add, Neg, Minus\}$. The two systems above are just two possible combinations of those features, but there are many other valid possibilities. Ideally, we would like to allow any possible combination of features, since in general it is not possible to know which of these features are relevant to the different clients. We expect the EFP to be particularly relevant in the context of component-based frameworks and software product-lines. In fact, the EFP is closely related to the *expression product lines* of Herrejon et al. [23]. Therefore, the final requirement for the EFP is that:

> *A solution should allow a high degree of composability and decoupling of components so that no valid combinations of features are ruled out.*

3 Extensible Encodings of Datatypes

In this section, we discuss the relationship between visitors and encodings of datatypes, and show how to make these encodings extensible. This will provide the foundations for the two Scala solutions presented in Sections 4 and 5.

3.1 Encodings of Datatypes and the Visitor Pattern

The Visitor design pattern [11] shows how to separate the structure of an object hierarchy from the behaviour of traversals over that hierarchy; it can be used in object-oriented languages to provide a functional decomposition style. Buchlovsky and Thielecke [7] formalized the relation between two variants of the Visitor pattern and encodings of datatypes in a minor variant of System F_ω with products. They observed that *external visitors* (visitors where the traversal of the object structure is explicitly controlled by the programmer) are related to Parigot encodings of datatypes [10], while internal visitors (visitors where the traversal is automatically performed by the object structure) are related to Church encodings of datatypes [9]. The basic idea behind the relationship between visitors and encodings of datatypes is briefly illustrated next (the reader wishing to know more details may look at [7, 8]):

$$Expr \equiv \forall X.\ \overbrace{\underbrace{(Int \Rightarrow X)}_{num} \Rightarrow\ \underbrace{(X \Rightarrow X \Rightarrow X)}_{add}}^{ExprVisitor} \Rightarrow X$$

$$
\begin{array}{ll}
\mathit{ExprVisitor}\ X & \equiv \{\, \mathit{num} \in \mathit{Int} \Rightarrow X, \mathit{add} \in X \Rightarrow X \Rightarrow X \,\} \\
\mathit{Expr} & \equiv \{\, \mathit{accept} \in \forall X.\ \mathit{ExprVisitor}\ X \Rightarrow X \,\} \\
\mathit{Num} & \in \mathit{Int} \Rightarrow \mathit{Expr} \\
\mathit{Num}\ x & \equiv \{\, \mathit{accept}\ v \equiv v.\mathit{num}\ x \,\} \\
\mathit{Add} & \in \mathit{Expr} \Rightarrow \mathit{Expr} \Rightarrow \mathit{Expr} \\
\mathit{Add}\ e1\ e2 & \equiv \{\, \mathit{accept}\ v \equiv v.\mathit{add}\ (e1.\mathit{accept}\ v)\ (e2.\mathit{accept}\ v) \,\}
\end{array}
$$

Fig. 1. Church encoding for numeric expressions using records.

$$
\begin{array}{ll}
\mathit{ExprVisitor}\ X & \equiv \{\, \mathit{num} \in \mathit{Int} \Rightarrow X, \mathit{add} \in \mathit{Expr} \Rightarrow \mathit{Expr} \Rightarrow X \,\} \\
\mathit{Expr} & \equiv \{\, \mathit{accept} \in \forall X.\ \mathit{ExprVisitor}\ X \Rightarrow X \,\} \\
\mathit{Num} & \in \mathit{Int} \Rightarrow \mathit{Expr} \\
\mathit{Num}\ x & \equiv \{\, \mathit{accept}\ v \equiv v.\mathit{num}\ x \,\} \\
\mathit{Add} & \in \mathit{Expr} \Rightarrow \mathit{Expr} \Rightarrow \mathit{Expr} \\
\mathit{Add}\ e1\ e2 & \equiv \{\, \mathit{accept}\ v \equiv v.\mathit{add}\ e1\ e2 \,\}
\end{array}
$$

Fig. 2. Parigot encoding for numeric expressions using records.

This example is based on the type of a Church encoding for a simple datatype of expressions. What the reader should note is that the two functional arguments *num* and *add* can be seen as, what in the VISITOR pattern are called, the *visit* methods for the type *Expr*. In order to make the connection to OO languages more clear we will assume, in what follows, a calculus much like the one presented by Buchlovsky and Thielecke, but also featuring subtyping [24] and using records [25] instead of products.

In Figure 1, instead of defining *Expr* as a higher-order function type, we use a record *ExprVisitor* to capture the visitor type and *visit* methods explicitly. We also use a record for *Expr* and name the functional type as *accept*. The two functions *Num* and *Add* are the two constructors (or concrete elements) for the *Expr* datatype. This is essentially an instance of the VISITOR pattern and can be easily translated into any OO language with support for generics.

A very similar construction can be done using Parigot encodings instead (but we need to additionally extend the calculus with both value and type level recursion). We show the code for Parigot encodings in Figure 2. The essential difference to Church encodings is that, for constructors with recursive occurrences of expressions such as *Add*, the expressions are not traversed by the constructor but are instead passed to the *add* visit method, delegating the responsibility of traversal to the client implementing the *add* operation.

Buchlovsky and Thielecke show that we can provide a *shape generic* version of the encodings that can be instantiated with different visitor shapes, by parametrizing over the visitor type — in this context "shape" essentially means

$$
\begin{array}{ll}
\mathit{Expr}\ V & \equiv \{\, \mathit{accept} \in \forall X.\, V\ X \Rightarrow X \,\} \\
\mathit{num}\ X & \equiv \{\, \mathit{num} \in \mathit{Int} \Rightarrow X \,\} \\
\mathit{add}\ \ X & \equiv \{\, \mathit{add}\ \ \in X \Rightarrow X \Rightarrow X \,\} \\
\mathit{Expr}_{\mathit{Num}}\ (V <: \mathit{num}) & \equiv \mathit{Expr}\ V \\
\mathit{Expr}_{\mathit{Add}}\ \ (V <: \mathit{add}) & \equiv \mathit{Expr}\ V \\
\mathit{Num} & \in \forall (V <: \mathit{num}).\ \mathit{Int} \Rightarrow \mathit{Expr}_{\mathit{Num}}\ V \\
\mathit{Num}\ x & \equiv \{\, \mathit{accept}\ v \equiv v.\mathit{num}\ x \,\} \\
\mathit{Add} & \in \forall (V <: \mathit{add}).\ \mathit{Expr}\ V \Rightarrow \mathit{Expr}\ V \Rightarrow \mathit{Expr}_{\mathit{Add}}\ V \\
\mathit{Add}\ e1\ e2 & \equiv \{\, \mathit{accept}\ v \equiv v.\mathit{add}\ (e1.\mathit{accept}\ v)\ (e2.\mathit{accept}\ v) \,\}
\end{array}
$$

Fig. 3. Extensible Church encoding using record subtyping.

the set of visit methods in a visitor. We need two versions of the shape generic encodings for internal and external visitors.

$\mathit{Internal}\ V \equiv \{\, \mathit{accept} \in \forall X.\, V\ X \Rightarrow X \,\}$

$\mathit{External}\ V \equiv \{\, \mathit{accept} \in \forall X.\, V\ (\mathit{External}\ V)\ X \Rightarrow X \,\}$

In each case, V is a *type constructor* (that is, a type that is itself parametrized by other types) and abstracts over the concrete visitor components. In the case of *Internal*, the visitor only needs to be parametrized by the result type. For *External*, the visitor also requires a second argument for abstracting over the recursive occurrences of *External*. Although type constructors are native to calculi of the System F_ω family, they are not normally found in mainstream OO languages with generics, since only first-order type parameters are allowed. So, these generic versions of visitors cannot be encoded in those languages. However, Scala has recently been extended with support for type constructors [13] and there have been proposals for supporting them in Java too [26].

3.2 Extensible Encodings of Datatypes using Record Subtyping

A major problem with the encodings of datatypes presented in Section 3.1 is that they are not extensible: we cannot easily add new variants to a datatype. With a standard encoding like the one presented in Figure 1, the datatype (or composite) type needs to know in advance about all the variants because of the fixed *shape* imposed by *ExprVisitor*. Interestingly, in the generic version of the encodings, the visitor shape is abstracted and the composite types *Internal* and *External* are not tied to any particular variants. Inspired by this observation, we can define an expression type that does not commit to a particular shape:

$\mathit{Expr}\ V \equiv \{\, \mathit{accept} \in \forall X.\, V\ X \Rightarrow X \,\}$

(This is basically the same type as *Internal*). We could easily obtain the type for expressions presented in Figure 1, by simply parametrizing *Expr* with *ExprVisitor*. However, we want to be able to define the constructors for numeric and addition expressions in a way that does not commit to a particular shape.

Clearly, we seek a solution that provides just enough information to define the constructor, but no more. In fact, all we need to know is that, for the constructor that we are defining, the visitor provides a corresponding visit method. This *minimal shape information* can be easily captured using standard record subtyping bounds as we can see in Figure 3. The type $Expr\ V$ is, as we have already discussed, just the type for expressions with a parametrized visitor shape. The types $num\ X$ and $add\ X$ define two atomic visitor components that provide, respectively, *num* and *add* visit methods. Here, we use the convention that these atomic visitor types have names spelled in exactly the same way than the visit methods they contain. The idea is that when we see a bound like $V <: num$ we can read it as "the visitor V contains the visit method *num*". The types $Expr_{Num}\ V$ and $Expr_{Add}\ V$ define refinements of $Expr\ V$ that specify some extra information about the shape. These types are used to provide constructors with more accurate types; but we should note that they are orthogonal to the extensibility problem and a slightly simpler extensible encoding can be achieved by just using $Expr\ V$ instead. Finally, the constructors *Num* and *Add* are defined almost in the same way as with traditional Church encodings. The only difference is that the types of our extensible encodings only assume minimal shape information by using subtyping bounds to specify which visitor component provides the respective visit method.

With this encoding the expression type is parametrized by a shape instead of having a hard reference to a particular shape, which decouples the expression type from the visitor. Furthermore, the constructors only need minimal shape information, which allows them to be developed independently of other variants. This means that adding new variants and new functions is possible and, consequently, achieves a solution to the expression problem. A very similar construction can be done for Parigot encodings. We will now switch to Scala and explore solutions to the expression (families) problem using both generic Church encodings (in Section 4) and generic Parigot encodings (in Section 5).

4 Modular Internal Visitor Components

In this section we explore a solution to the expression families problem using modular internal visitors, inspired by Church encodings of datatypes.

4.1 Modular Internal Visitors in Scala

In Figure 4 we show a translation of the code in Figure 3 into Scala. Apart from fairly obvious idiomatic conversions (like, for example, encodings types as *traits* and *classes*) the Scala code is surprisingly faithful to the original code in Figure 3. Even though there is a significant gap between a calculus like System $F^{<:}_{\omega}$ and Scala, the fact is that Scala supports the essential features that are required by the encodings. In particular, the encoding requires *type parametrization* (or *parametric polymorphism*) in both the first-order and higher-order forms, the

```
trait Expr[-V[_]] {
  def accept[a] (vis : V[a]) : a
}
trait num[A] {
  def num (x : Int) : A
}
case class Num[-V[X] <: num[X]] (x : Int) extends Expr[V] {
  def accept[a] (vis : V[a]) : a = vis.num (x)
}
trait add[A] {
  def add (e1 : A, e2 : A) : A
}
case class Add[-V[X] <: add[X]] (e1 : Expr[V], e2 : Expr[V]) extends Expr[V] {
  def accept[a] (vis : V[a]) : a = vis.add (e1.accept (vis), e2.accept (vis))
}
```

Fig. 4. Extensible expressions in Scala.

latter of which has been recently added to Scala [13]. The most significant difference between the Scala version and the System $F^{<:}_{\omega}$ version is the use of a *contravariance* annotation (the "-" preceding V) for the visitor type parameter. This annotation is not strictly necessary, but without it this solution would not preserve the following subtyping relationship

$Expr[V] <: Expr[U]$ **if** $U <: V$

which is one of the requirements for a solution for the EFP. There are a few other minor points that are worthwhile noting. Firstly, the Scala version combines the definitions of the constructors with the refined types for those constructors. For example, in the extensible Church encoding, we define a type $Expr_{Num}$ which captures the more refined type for the result type of the constructor Num. In Scala, a class declaration together with the **extends** clause captures these two constructions. Secondly, in Scala type constructor declarations are provided together with their corresponding arity and bounds. For example, $V\ [X] <: num[X]$ declares a type constructor variable V that has one type argument X and is bounded by $num[X]$. In the definition $Expr[-V\ [\_]]$, naming the type constructor argument is not necessary, so we use the anonymous variable "_" to declare the existence of one type argument. Finally, we use a *case class* [27] instead of a standard class for syntactical brevity when constructing new values (since it allows us to avoid uses of **new**).

4.2 Adding New Operations

An operation that evaluates expressions can be defined, using a visitor, with the following trait:

```
trait BaseEval extends num[Int] with add[Int] {
  def num (x : Int)            = x
  def add (e1 : Int, e2 : Int) = e1 + e2
}
```

This trait extends the numeric and addition visitors, using mixin composition [14] of traits, and provides the definition for the corresponding visit methods. Because we use an internal visitor, all the traversal code is handled in the constructors, so in the *add* visit method, the only thing that is left to be done is to add the two results together.

We can write some simple testing code that demonstrates a possible way to use *BaseEval* from a client perspective.

```
type numadd[A] = num[A] with add[A]
type NumAdd    = Expr[numadd]

def exp : NumAdd =
  Add[numadd] (Num[numadd] (3), Num[numadd] (4))

def evalNumAdd (e : NumAdd) : Int = e.accept (new BaseEval () {})

val test1 : Int = evalNumAdd (exp)
```

For the sake of clarity and brevity, we define *numadd* and *NumAdd* type synonyms, which correspond, respectively, to the visitor and composite types instantiated with a more concrete shape. We create a basic test expression *exp* that encodes the expression $3+4$ and test it by calling the *evalNumAdd* on that expression. There are a couple of inconveniences about this client code that we should note. Firstly, we need to parametrize the constructors with the visitor type, which makes the use of constructors significantly verbose (we would like to write *Add* (*Num* (3), *Num* (4)) instead). Secondly, we are providing *evalNumAdd* in the client code. It would be preferable to have a "generic" *eval* definition that would be provided in the library code instead. We shall address these convenience issues in Section 6.

4.3 Adding New Variants and Extending Existing Operations

Suppose that we want to add a new constructor that negates expressions. With our approach, this is also very easy: all we need to do is to introduce the visitor and corresponding constructor.

```
trait neg[A] {
  def neg (e : A) : A
}
case class Neg[-V[X] <: neg[X]] (e : Expr[V]) extends Expr[V] {
  def accept[a] (vis : V[a]) : a = vis.neg (e.accept (vis))
}
```

The trait *neg* is the visitor type and defines the *neg* visit method and the case class *Neg* defines a constructor taking a single expression as argument.

We can provide a definition for *eval* independently of the definitions for *num* and *add*

```
trait NegEval_1 extends neg[Int] {
  def neg (e : Int) = −e
}
```

and later mix it in with those definitions:

```
trait NumAddNegEval extends BaseEval with NegEval_1
```

Alternatively, we could directly extend *BaseEval*:

```
trait NegEval_2 extends BaseEval with neg[Int] {
  def neg (e : Int) = −e
}
```

4.4 Subtyping Between Components for Scalability and Reuse

Interestingly, while we may think that the trait $NegEval_1$ is more reusable than $NegEval_2$ (since it has no references to *BaseEval*) *this is, in fact, not the case*! Indeed the two variants are equally reusable and there is no advantage of one against the other in that respect. Because visitor extension usually follows the standard subtyping relation (although there are some exceptions, as shown in Section 4.5), a concrete visitor supporting *num*, *add* and *neg* can be passed when a visitor just supporting *num* and *add* is expected. For example, we could have alternatively defined *evalNumAdd* in the client code as:

```
def evalNumAdd (e : Expr[numadd]) : Int = e.accept (new NegEval_2 () {})
```

The point here is that we do not need to carefully design visitor components for operations like this one independently of each other, which is helpful for scalability: we can pack many cases together (like in the trait *BaseEval*) and avoid code scattering and redundancy.

Another interesting point that is worthwhile noting is that, because of the subtyping relationships between different types of expressions we can apply operations defined over some type of expressions to expressions with strictly fewer variants. For example,

```
def evalNumAddNeg (e : Expr[numaddneg]) = e.accept (new NegEval_2 () {})
val test_2 = evalNumAddNeg (exp)
```

the function *evalNumAddNeg* takes an expression that supports numeric, addition and negation variants, but *exp* (defined above) is a *different* type of expressions that supports numeric and addition variants only. However, because $Expr[numadd] <: Expr[numaddneg]$ we can pass *exp* to *evalNumAddNeg*.

4.5 Narrowing Operation

As we pointed out in Section 2 a solution to the EFP should allow the incremental definition of a narrow operation, so that it can be reused by any pair of expression types. With our solution we can achieve this by creating a visitor component that is itself parametrized by the type of another visitor component (which is the shape of the target expression type). We show the code for the

```
trait NumNarrow[V[X] <: num[X]] extends num[Expr[V]] {
  def num (x : Int)                    = Num[V] (x)
}
trait AddNarrow[V[X] <: add[X]] extends add[Expr[V]] {
  def add (e1 : Expr[V], e2 : Expr[V]) = Add[V] (e1, e2)
}
trait NegNarrow[V[X] <: neg[X]] extends neg[Expr[V]] {
  def neg (e : Expr[V])                = Neg[V] (e)
}
trait NMNarrow[V[X] <: num[X] with minus[X]] extends neg[Expr[V]] {
  def neg (e : Expr[V])                = Minus[V] (Num[V] (0), e)
}
```

Fig. 5. Components for the narrow operation.

narrow components in Figure 5. We expect that, for the most part, the majority of the variants are shared between the two expression types involved in the narrow operation and that the conversion between those variants will essentially be a matter of decomposing the variant of the input expression, narrowing recursively and rebuilding the same variant on the output expression. The visitors *NumNarrow*, *AddNarrow* and *NegArrow* do exactly this. However, when the target type of the expression does not have the variant that we are interested in, we need to convert the expression using some other variants. The *NMNarrow* visitor shows how we could provide an alternative translation from an expression with negation into an expression without that variant, by using numeric and subtraction variants (we assume the existence of the visitor *minus* and the *Minus* variant here). Note that the following definition for *neg*

```
def neg (e : Expr[V]) = Neg[V] (e)
```

would be a static type error in the *NMNarrow* trait. By using mixin composition, we are free to assemble a narrow operation in very flexible ways and there may be multiple alternatives to pick from for the same case. For example, the object

```
object myNarrow extends NumNarrow[num] with NMNarrow[numminus]
```

provides a concrete narrow visitor that converts between expressions with *Num* and *Neg* variants into expressions with *Num* and *Minus* variants. Unlike the visitor for evaluation, with the narrow operation visitors we need to be careful when grouping the different cases together since we can create dependencies on variants because of the constraints imposed by the visitor type argument.

5 Modular External Visitor Components

In this section we explore a solution to the expression families problem using modular external visitors, inspired by Parigot encodings of datatypes.

```
object Components {
  // The base component for expression families
  trait Expr[−V[−_, _]] {
      def accept[a] (vis : V[Expr[V], a]) : a
  }
  // The components for the Num variant
  trait num[−R, A] {
      def num (x : Int) : A
  }
  case class Num[V[−R, A] <: num[R, A]] (x : Int) extends Expr[V] {
      def accept[a] (vis : V[Expr[V], a]) : a = vis.num (x)
  }
  // The components for the Add variant
  trait add[−R, A] {
      def add (e1 : R, e2 : R) : A
  }
  case class Add[V[−R, A] <: add[R, A]] (e1 : Expr[V], e2 : Expr[V])
    extends Expr[V] {
      def accept[a] (vis : V[Expr[V], a]) : a = vis.add (e1, e2)
  }
  // The components for the Neg variant
  trait neg[−R, A] {
      def neg (e : R) : A
  }
  case class Neg[−V[−R, A] <: neg[R, A]] (e : Expr[V]) extends Expr[V] {
      def accept[a] (vis : V[Expr[V], a]) : a = vis.neg (e)
  }
  // An evaluation component
  trait EvalVisitor[V[−R, A]] extends
    num[Expr[V], Int] with add[Expr[V], Int] with neg[Expr[V], Int] {
      self : V[Expr[V], Int] ⇒
      def num (x : Int)                    = x
      def add (e1 : Expr[V], e2 : Expr[V]) = e1.accept (this) + e2.accept (this)
      def neg (e : Expr[V])                = −e.accept (this)
  }
  // Some components for the narrow operation
  trait NumNarrow[V1[−_, _], V2[−R, X] <: num[R, X]]
    extends num[Expr[V1], Expr[V2]] {
      def num (x : Int)                    = Num[V2] (x)
  }
  trait AddNarrow[V1[−_, _], V2[−R, X] <: add[R, X]]
    extends add[Expr[V1], Expr[V2]] {self : V1[Expr[V1], Expr[V2]] ⇒
      def add (e1 : Expr[V1], e2 : Expr[V1]) =
        Add[V2] (e1.accept (this), e2.accept (this))
  }
}
```

Fig. 6. The library code for expression components.

5.1 Modular External Visitors in Scala

In Figure 6 we show the Scala code necessary to implement a small library of expression components using modular external visitors. The trait *Expr* defines the base component for our expression families; all constructors extend this trait. Like with the internal visitor solution, we need a contravariance annotation for the visitor type parameter $V\,[-\_, \_]$. However, we also need an extra contravariance annotation for the first type argument of V. As before, these variance annotations are required to ensure that the following subtyping relation holds:

$Expr\,[\,V\,] <: Expr\,[\,U\,]$ **if** $U <: V$

but, if we did not want to preserve this relation, then the contravariance annotation would not be required. Visitors take two type arguments instead of a single one (when compared to the internal visitor solution) because we need to distinguish the types of the recursive arguments from the result type.

We provide three variants in the library for numeric, addition and negation expressions. The constructors define *accept* methods that do not recur on the expressions, delegating that responsibility to the visitors, and following the Parigot encoding of datatypes presented in Figure 2. Two sets of components for operations are provided: the first one evaluates expressions; and the second one provides some definitions for the narrow operation. For operations with recursive calls we need a self-type annotation because, without the annotation, it would not be safe to assume that all the cases present in the expressions being recursively traversed would be handled. This is the same issue that was encountered, for example, by Torgersen [18] in his second solution for the expression problem.

In this section, we do not provide a step-by-step explanation of how independent extensibility of components can be achieved, because this can be done in essentially the same way as the solution presented in Section 4. We focus instead on discussing some practical concerns when assembling components and also on the extra expressiveness provided by external visitors over internal visitors.

5.2 Ad-hoc Assembling of Components

The code presented in Figure 6 captures the code involved in a family of expressions, but it does not define any member of that family in particular. We need to *combine* (some of the) expression components if we want to obtain a particular type of expressions. The combination of components is not a responsibility of the library writer, because he cannot predict which combinations are interesting. Obviously, he cannot enumerate all possible combinations too, since the number of combinations rises very fast in respect to the number of components. So, *the assembling of components should be delegated to the clients of the library.*

In Figure 7 we present the code for a client of the component library, which supports expressions with numeric and addition variants and evaluation. The value C is used as a shortcut to the *Components* object (note that, in Scala, objects also play the role of modules). The type *ExprShape* defines a concrete visitor shape that combines several smaller visitors using mixin composition; and then we use that shape to define the type of expressions *Expr*. We also

```
trait Client {
  protected val C = Components
  //Defining the members of the datatype
  protected type ExprShape[−R, A] = C.num[R, A] with C.add[R, A]
  type Expr                       = C.Expr[ExprShape]
  //Shorthand for Expression Visitors
  trait ExprVisitor[A] extends C.num[Expr, A] with C.add[Expr, A]
  //Shorthands for the constructors
  def Num (x : Int)               = C.Num[ExprShape] (x)
  def Add (e1 : Expr, e2 : Expr) = C.Add[ExprShape] (e1, e2)
  //The operations
  def eval (e : Expr) : Int =
    e.accept[Int] (new C.EvalVisitor[ExprShape] () {})
}
```

Fig. 7. Ad-hoc assemblage of components for expressions.

define an *ExprVisitor* trait that can be used to easily create new visitors for our expressions. Next we define some useful shorthands for the constructors, which avoid parametrization over the visitor type. Finally, operations like *eval* are defined by calling the *accept* method on the corresponding visitor.

The nice thing about this client is that it provides an abstraction on top of the component library. This is important because the components of the library use some advanced Scala features and extra parametrization that would not normally be needed if the program had been defined conventionally. If those components had been used directly, then some familiarity with the Scala features used in the library would probably be needed and difficult to interpret error messages arising from the misuse of these features would almost certainly occur. Happily, any code that uses *Client* does not need to be aware of the components in the expression library: all that is visible is a fairly *conventional* interface. However, the definition of clients like this one is somewhat ad-hoc, and similar preparation code is needed for other clients. In Section 6, we show how we can define these client interfaces in a more compositional and less ad-hoc way.

5.3 Extensible Modular Components with Multiple Dispatching

As the reader may notice, external visitors are more complicated to use than internal visitors because they require extra typing and the responsibility of traversal is delegated to the programmer. So, an obvious question is why should we bother with external visitors in the first place. Ignoring the extensibility issue for a moment, the main reason to use external visitors is when the recursion pattern of the operations we are defining does not follow a simple *structural recursion*, which is what internal visitors excel at. External visitors are essentially equivalent to *case analysis* [8] and, in a language like Scala, they can be used

to define operations that do not follow standard recursion patterns. In particular, with external visitors it is possible to define operations that *dynamically dispatch over multiple arguments* or perform *nested case analysis* over some of the arguments.

The interesting question to ask is whether the ability to define these non-standard recursive schemes translates into our modular external visitors. This would imply a modular and statically type-safe solution for *extensible* multiple dispatching, without the need for any special purpose language extensions. As we shall see, this is indeed possible, but it is not simple. The good news is that there is a fairly mechanical scheme that can be used to define operations with such recursion patterns, which hints at a possible higher-level notation similar to multi-methods [28, 19] or pattern matching as a language extension.

We use structural equality between expressions (which is a binary method) as our working example. When working with non-extensible visitors, the trick to achieve multiple dispatching is to use a series of visitors to handle each dispatching (the reader may look at [8] for an example of *equality* defined in this way). The strategy that we will use to define extensible equality is similar. It is helpful to look at a definition of equality by pattern matching to understand what happens when we define the modular components for structural equality:

$$
\begin{aligned}
&eq :: (Expr, Expr) \to Expr \\
&eq\ (Num\ n1, Num\ n2) &&= n1.equals\ (n2) \\
&eq\ (Add\ e1\ e2, Add\ e3\ e4) &&= eq\ (e1, e3) \wedge eq\ (e2, e4) \\
&eq\ (Neg\ e1, Neg\ e2) &&= eq\ (e1, e2) \\
&eq\ (\_, \_) &&= false
\end{aligned}
$$

There is some modularity in a definition like this. In order to add a new clause, we do not need to touch the code of other clauses. We explore exactly the same form of modularity in our components for equality shown in Figure 8. The *BaseHandleDefault* visitor, handles the default cases that return false. This can be seen as the code corresponding to the last clause in the definition of *eq*. In order to handle one of the other clauses we need three visitors: one for extending the default visitor with the new case, another for handling the first matched pattern and a third one to handle the second matched pattern. For the *eq* (*Num n1*, *Num n2*) clause, the *NumHandleDefault* visitor extends the default visitor with a *num* visit case. The *NumEquals* visitor defines the case for the first matched pattern and calls an instance of the visitor than handles the second match, which is handled by the third visitor *HandleNum*. Providing code for other clauses proceeds in a similar fashion. We show the code that handles the *eq* (*Neg e1*, *Neg e2*), but skip the code for *eq* (*Add e1 e2*, *Add e3 e4*) for space reasons.

6 Feature-Oriented Programming

In this section, inspired by ideas from *feature-oriented programming* (FOP) [29], we show how to organize components into features that can be used to easily and *compositionally* assemble customized expressions datatypes and operations.

```
object ExtendedComponents {
  //Default case for equality : eq (_, _) = false
  trait BaseHandleDefault[V[−_, _], A] {
      self : V[Expr[V], A ⇒ Boolean] ⇒
      //recursive call reference
      def eqVis : V[Expr[V], Expr[V] ⇒ Boolean]
      //default value
      val default = (_ : A) ⇒ false
  }
  //Components for handling : eq (Num n1, Num n2) = n1.equals (n2)
  trait NumHandleDefault[V[−_, _], A] extends BaseHandleDefault[V, A]
    with num[Expr[V], A ⇒ Boolean] {
      self : V[Expr[V], A ⇒ Boolean] ⇒
      def num (n2 : Int) = default
  }
  trait NumEquals[V[−_, _]] extends num[Expr[V], Expr[V] ⇒ Boolean] {
      self : V[Expr[V], Expr[V] ⇒ Boolean] ⇒
      def eqNum : V[Expr[V], Int ⇒ Boolean]
      def num (n : Int) = e ⇒ e.accept (eqNum) (n)
  }
  trait HandleNum[V[−R, A]] extends NumHandleDefault[V, Int] {
      self : V[Expr[V], Int ⇒ Boolean] ⇒
      override def num (n2 : Int) = n1 ⇒ n1.equals (n2)
  }
  //Components for handling : eq (Neg e1, Neg e2) = eq (e1, e2)
  trait NegHandleDefault[V[−_, _], A] extends BaseHandleDefault[V, A]
    with neg[Expr[V], A ⇒ Boolean] {
      self : V[Expr[V], A ⇒ Boolean] ⇒
      def neg (e : Expr[V]) = default
  }
  trait NegEquals[V[−R, A]] extends neg[Expr[V], Expr[V] ⇒ Boolean] {
      self : V[Expr[V], Expr[V] ⇒ Boolean] ⇒
      val eqNeg : V[Expr[V], Expr[V] ⇒ Boolean]
      def neg (e2 : Expr[V]) = e1 ⇒ e1.accept (eqNeg) (e2)
  }
  trait HandleNeg[V[−R, A]] extends NegHandleDefault[V, Expr[V]] {
      self : V[Expr[V], Expr[V] ⇒ Boolean] ⇒
      override def neg (e2 : Expr[V]) = e1 ⇒ e1.accept (eqVis) (e2)
  }
}
```

Fig. 8. Extensible components for equality.

6.1 Organizing Components into Features

In Section 5.2 we have already seen how we can fairly easily assemble visitor components in an ad-hoc, non-compositional way. However, some overhead is still required. Ideally, assembling a final system should be as easy as composing a few smaller subsystems together. The comments in Figure 6 identify what components are needed for *numeric*, *addition* and *negation* variants and which components are needed for *evaluation* and *narrowing*. Each of these groups of components can be seen as what in FOP is called a *feature*.

In Scala it is possible to more precisely capture these features by grouping the required functionality for each feature in a trait. We illustrate this in Figure 9. The *Base* feature (on which all other features depend) abstracts over the concrete visitor shape using a virtual type *ExprVisitor*, and a type *Expr* defines the type of expressions with that shape. Note that we could also have parametrized *Base* by the visitor instead of using an abstract type, but we feel that an abstract type captures the nature of the abstraction better here. The *Numeric* feature imposes a constraint on the shape in order to support the *num* visit method, and defines a method *Num* that can be used to construct numeric expressions with the particular shape required by *ExprVisitor*. The features for *Addition* and *Negation* are defined in a similar way to *Numeric*, imposing corresponding constraints on the visitor shape and defining a constructor method. The *Eval* feature defines a type *EvalVisitor* that specifies the expected type for evaluation visitors for the particular *ExprVisitor* shape. A method *eval* that supports evaluation of expressions is also specified in the trait by using an instance of *EvalVisitor*. However, this instance reference is abstract (because it cannot be created without knowing the final shape) and is expected to be provided in the object implementing the trait. The narrowing feature requires a second abstract visitor, which defines the shape of the output expression type for the narrowing operation. The *NumNarrow* and *NegNarrow* traits are examples of *composite features* (that is, they are built on top of more basic features). Each of these two features constrains the output visitor type of the narrowing operation.

Figure 10 shows how we could assemble a client by combining some of the features using mixin composition. The first line of the object declaration for *Client* expresses what we may expect from a FOP language, defining a client to be the composition of three features that will provide support for numeric and subtraction variants together with evaluation and a narrowing operation. In Scala we still need to do a little bit more work because we need to instantiate the visitor shapes and the visitors required for the operations, but this is fairly trivial code and certainly shorter than the code that needs to be provided for a client like the one in Figure 7.

7 Related Work

In this section we discuss related work. We also assess existing solutions to the extensibility problem against the requirements of the EFP.

```
trait Base {//Base feature
  protected val C = Components
  protected type ExprVisitor[−R, A]
  type Expr = C.Expr[ExprVisitor]
}
trait Numeric extends Base {//Numeric Feature
  type ExprVisitor[−R, A] <: C.num[R, A]//feature constraints
  def Num (x : Int) = C.Num[ExprVisitor] (x)
}
trait Addition extends Base {//Addition Feature
  type ExprVisitor[−R, A] <: C.add[R, A]//feature constraints
  def Add (e1 : Expr, e2 : Expr) = C.Add[ExprVisitor] (e1, e2)
}
trait Negation extends Base {//Negation Feature
  type ExprVisitor[−R, A] <: C.neg[R, A]//feature constraints
  def Neg (e : Expr) = C.Neg[ExprVisitor] (e)
}
trait Eval extends Base {//Evaluation Feature
  protected type BaseEval    = C.BaseEval[ExprVisitor]
  protected type EvalVisitor = BaseEval with ExprVisitor[Expr, Int]
  protected val evalVisitor : EvalVisitor//abstract
  def eval (e : Expr) : Int = e.accept[Int] (evalVisitor)
}
trait Narrow extends Base {//Narrowing Feature
  type TExpr = C.Expr[TExprVisitor]
  protected type TExprVisitor[−R, A]
  protected type NarrowVisitor = ExprVisitor[Expr, TExpr]
  protected val narrowVisitor : NarrowVisitor//abstract
  def narrow (e : Expr) : TExpr = e.accept[TExpr] (narrowVisitor)
}
trait NumNarrow extends Numeric with Narrow {//Narrowing for numbers
  protected type TExprVisitor[−R, A] <: C.num[R, A]
  protected type NumNarrow = C.NumNarrow[ExprVisitor, TExprVisitor]
}
trait NegNarrow extends Negation with Narrow {//Narrowing for negation
  protected type TExprVisitor[−R, A] <: C.num[R, A] with C.minus[R, A]
  protected type NMNarrow = C.NMNarrow[ExprVisitor, TExprVisitor]
}
```

Fig. 9. Expression features.

```
object Client extends NumNarrow with NegNarrow with Eval {
  type TExprVisitor[−R, A]    = C.num[R, A] with C.minus[R, A]
  type ExprVisitor[−R, A]     = C.num[R, A] with C.neg[R, A]
  protected val evalVisitor   = new BaseEval {}
  protected val narrowVisitor = new NumNarrow with NMNarrow
}
```

Fig. 10. A client with numeric, negation, narrowing and evaluation features.

7.1 Extensible Visitors and Algebraic Datatypes

There have been several proposals to make visitors more flexible and extensible in the past [30–32]. Like our solution, an important motivation for most of these approaches is to remove the dependencies between visitors and concrete subclasses of the object structure. As Vlissides [32] observes, the VISITOR pattern (in its classic form) is unsuitable to be used in frameworks because of the references to concrete subclasses, violating the dependency inversion principle [33] and endangering modularity. However, the flexibility and extensibility in those approaches comes at a price: the solutions are not statically type-safe; casts or reflection are used and run-time type errors can occur if a visitor (or *visit* method) is called on a variant it does not handle. Both Krishnamurthi et al. [30] and Vlissides [32] describe variations of the VISITOR pattern that follow a structure similar to ours. The former solution can avoid run-time errors if all existing visitors are subclassed and some factory methods are overriden when a new variant is added; while the later solution can use catch-all cases for the same purpose. In both approaches the correct usage of the pattern (so that it does not incur of run-time type errors) is quite complex and error-prone.

Zenger and Odersky [20] propose *extensible algebraic datatypes with defaults* (EADDs) as a possible solution for the expression problem. They observe that the subtyping relationship between a datatype and its extension is inverted (the extension is a supertype of the original datatype), which leads to the idea of adding a default variant to every algebraic datatype. This has the effect of subsuming all variants defined in future extensions. Unlike our datatypes, in their approach *the extension is a subtype* of the original datatype. Because of this static type-safety is guaranteed even when a new, unforeseen, variant is added. However, this solution is subject to single inheritance and only linear extensions are possible. Moreover, it assumes that sensible default cases exist for all functions, which may not necessarily be the case. *Case classes* in Scala [27] and the *open datatypes and functions* proposal for Haskell [21] can be seen as close relatives of EADDs as they allow easy introduction of new variants and it is possible to provide a default case in a function, which ensures that the function will not fail with a run-time type error. Still, the use of a default case is not enforced, which provides some extra flexibility but also means that run-time type errors can occur.

One important requirement of the expression families problem (but not of the expression problem) is that *expressions used in different domains should have distinct types*. While most of the solutions above do solve the extensibility problem (even if at the cost of static type-safety), they do not easily allow us to have distinct types with reuse because we normally have single, simple types like *Expr*, *Num* or *Add* which are impossible to distinguish when used in different domains. Our solution allows the two distinct numeric expressions to have distinct types, while reusing most of the common, domain-independent functionality because we have types parametrized by visitors: $Expr\,[\,V\,]$, $Num\,[\,V\,]$ or $Add\,[\,V\,]$. In a sense, the visitor parameter can be seen as the different domain of expressions. So, by using two different visitor types we can distinguish between expressions used in different domains while achieving reuse.

7.2 Multiple Dispatch and Open Classes

Mainstream object-oriented languages, like C++, C# and Java, all use a *single dispatching* mechanism, where a single argument (the **self** object) is *dynamically* dispatched and all other dispatching is static. A problem arises, however, when a method requires dynamic dispatching on two or more arguments. The Visitor pattern can be seen as a way to emulate double-dispatching in a single dispatching language [34, 11]. By using nested visitors, we can also emulate a limited, non-extensible form of multiple dispatching. Modular visitors overcome the extensibility limitation and can be used to develop extensible and modular operations that dynamically dispatch over more than one argument. However, the use of visitors to emulate multiple dispatching is not trivial and, admittedly, it is much less practical to use than programming language extensions like multi-methods [28, 19, 35].

In a language with multiple dispatching the need for the classic Visitor pattern is greatly reduced as most multiple dispatching languages support the notion of *open classes* [19], since multi-methods are normally defined independently of the classes. Consequently, we can use multi-methods to add a new operation to an object structure modularly. However, this does not solve the problem of reuse across similar object structures while allowing distinct type identities (see the discussion at the end of Section 7.1). We believe that the two lines of work are essentially complementary. On the one hand, modular visitors could benefit from a mechanism similar to multi-methods or pattern matching to better express reusable, extensible and modular operations that dynamically dispatch over multiple arguments. There is an extensive amount of work around multi-methods covering syntax, type checking and ambiguities between different clauses; this could be very useful for such a hypothetical extension. On the other hand, our work could potentially provide an alternative compilation model for multi-methods targeting conventional single dispatching languages without using any form of run-time type analysis and while supporting modularity and extensibility. It would be interesting to explore this in the future.

7.3 Generics

Wadler proposed a solution using generics to solve the expression problem [5], but he later found a subtle typing problem. Kim Bruce [36] proposed a solution to the expression problem using generics and self-types. He also made an attempt to solve the expression problem using an instance of the VISITOR pattern (again with generics and self-types). However, he failed to obtain a fully statically type-safe visitor solution. Nevertheless, he observed that type constructors (that we use in our solution) could be useful. Torgersen's second and third solutions to the expression problem [18] addressed the typing problems of Bruce's solution and showed fully statically type-safe solutions just using conventional generics and an instance of the VISITOR pattern. The idea is simple: use imperative instead of functional style visitors. Consequently, visitors do not need to be parametrized types and type constructors can be avoided. Self-types are also avoided by parametrizing the *visit* methods with an extra visitor parameter provided by the concrete elements. These solutions are a close relative to the modular external visitors presented in Section 5. However, by avoiding type-constructors some expressiveness is lost. For example, it is no longer possible to apply the same technique to datatypes that are themselves parametrized by types (that is, types like $Vector\langle A\rangle$) as this would require visitors themselves to be parametrized by types. Furthermore, these solutions only work in languages with mutable-state, while functional-style visitors do not have such requirement. Torgersen also presented two other solutions for the problem: the first one works in both Java and C#, while the fourth relies on dynamic reification of type parameters that is only present in C#.

All of the generics solutions have an important characteristic in common with our solution: they are parametrized by the family of expressions or the family of visitors (or both). This means that, like our solution, it is possible to distinguish between different types of expressions. The third solution by Torgerson has another thing in common with our solution: the subtyping relationships between different expressions are preserved. An important limitation of these techniques is their lack of support for independent extensibility [6].

7.4 Type Classes and Polymorphic Variants

Oliveira et al. [37] addressed the problem of *extensible generic functions* in Haskell using records in the form of constructor type classes (that is, type classes parametrized by a type constructor) and noted the connection to the expression problem. This solution is essentially an instance of internal visitors in disguise [38] and inspired the solution presented in Section 4. Swierstra [39] proposed a solution to the expression problem using extensible sums (or variants) that has some close similarities to Oliveira's et. all technique and the solution proposed here. However, this approach relies on variant subtyping, which needs to be encoded in Haskell. From an OO perspective, Swierstra's technique seems less appealing than a solution that uses records because while nearly all OO languages natively support some form of record subtyping, most (if not all) do

not support variant subtyping and a manual implementation of the subtyping machinery for variants would also be required.

Garrigue [40] shows how *polymorphic variants* can be used to solve the expression problem. With polymorphic variants, different datatypes can share the same constructor. When a definition using pattern matching is written every usage of a polymorphic variant will raise a type constraint which ensures that only a datatypes containing all of those constraints will be used in the definition. An important drawback of this approach is that functions are not extensible and open recursion has to be used manually to emulate extensible functions.

Both the Haskell solutions and polymorphic variants have very good support for type inference. This can be seen as an advantage because it allows us to program without ever closing extensions. In our approach this is also possible but, because support for type inference in Scala is weaker, this becomes more cumbersome (see, for example, the client code in Section 4.2). However, by programming in this open style, the client will also be exposed to the complexity of the advanced language features to achieve extensibility, which can lead, for example, to difficult error messages to interpret. With our solution we can provide an abstraction on top of the reusable infrastructure that hides that complexity away. We believe that in practice having this abstraction is preferable as this keeps the interfaces very simple and familiar to most programmers. Also, in all these approaches there are important limitations when the functions we want to write do not follow a simple structurally recursive scheme.

7.5 Virtual Types

Odersky and Zenger [6] present two solutions for the expression problem using a combination of virtual types and nested classes. In the top-level classes, some operations and variants are initially added and the hard references that would preclude extensibility are replaced with virtual types. In the subclasses, new operations and/or variants can be added by suitably extending the top-level class and refining the virtual types. Their solution has, somehow, the flavour of *virtual classes*, which provide a more direct way to solve the problem as Ernst demonstrates in GBeta [41]. Ernst's solution also benefits from a special form of composition that can compose two classes and all of their inner classes automatically. In Scala we have to perform this operation manually. Nystrom et al. propose Java extensions similar to virtual classes that support *nested inheritance* [42] and *nested intersection* [43]; and present a solution for the EP that is very similar to the virtual classes solution by Ernst. More recently, Qi and Myers [44] have proposed *class sharing* as a new language mechanism that aims at allowing objects of one family to be used as members of another family. Our use of variance annotations to allow subtyping relations across components of different families also achieves this kind of interchange of objects in different families. Nonetheless, class sharing does not induce subtyping relations and can be used to make adaptations that are not possible with our approach. However, class sharing requires significant annotations, which places an additional burden on the programmer.

Solutions that use some form of virtual types (or classes) are generally very readable and easy to understand because the reusable code is very similar to the code that would be written if we would not be aiming at extensibility. In solutions like ours, or the ones discussed in Sections 7.3 and 7.4, the reusable code has to be written in a slightly different style and genericity becomes explicitly visible due to some extra typing effort involved. We believe that virtual types provide a particularly good solution to problems where a relatively small amount of customization is expected and a small, interesting set of composable functionality is identified. However, we think that when the expected degree of customization is higher and potentially all valid combination of features should be allowed, then virtual types do have some drawbacks. If we want to use virtual types to allow the degree of compositionality and decoupling required by the EFP, we basically need to have a class with the corresponding nested virtual types for each feature. Furthermore, we need to scatter the reusable code for the operations very finely across those classes so that entanglement between features is not created. Therefore, although it would be possible to achieve a similar degree of customization and compositionality, the readability advantage would be lost and a considerable amount of boilerplate code to set up each feature would be required. Moreover, if the language that we use supports virtual types, but not nested inheritance (like, for example, Scala) then the amount of effort to compose features can be quite overwhelming. Our solution on the other hand, allows small features to be created with very little boilerplate code and, for most operations, we do not need to scatter code around since, as we have discussed in Section 4.4, we can exploit the subtyping relationship between visitors to group many cases together without entangling features.

8 Conclusions

We have shown how to solve the EFP using two alternative variations of modular visitors. One very simple and practical alternative is to use internal visitors. Another alternative is to use external visitors, which are slighty more complex to use but allow additional expressiveness. Inspired by some ideas of FOP, we have also shown how to organize the visitor components into features that can be easily composed to provide customized systems of datatypes and operations. We believe that our techniques can be very helpful for the development of software in a FOP style without requiring any special tool or language extension and using only generic language constructs.

While in most situations internal visitors are preferable, there are a few situations where external visitors may be more suitable, which seems to force us into a design decision. In earlier work [8] we have presented a reusable, generic and type-safe visitor library (VisLib) that is parametrizable over the traversal strategy. Internal and External visitors can be recovered by suitably parametrizing the concrete visitors with the corresponding traversal strategy. As it happens, *extensibility is orthogonal to VisLib* and we can in fact easily use the original VisLib to develop extensible visitor components using techniques similar to the

ones in this paper, without having to commit to internal or external visitors in advance. Although we have not presented such solution here, in the companion code for this paper a solution using VisLib is also presented and documented.

Acknowledgements

We are very thankful to the anonymous reviewers for their excellent reviews, which have greatly helped to improve the presentation of this paper. Jeremy Gibbons provided valuable feedback on an earlier draft. This work is supported by the EPSRC grant *Generic and Indexed Programming* (EP/E02128X).

References

1. McIlroy, D.: Mass produced software components. [2] 138–155
2. Naur, P., Randell, B., eds.: Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany. (1969)
3. Reynolds, J.C.: User-defined types and procedural data structures as complementary approaches to type abstraction. In Schuman, S.A., ed.: New Directions in Algorithmic Languages, Rocquencourt (1975) 157–168
4. Cook, W.R.: Object-oriented programming versus abstract data types. In: REX Workshop/School on the Foundations of Object-Oriented Languages. LNCS 173, Springer-Verlag (1990) 151–178
5. Wadler, P.: The expression problem. Java Genericity Mailing list (November 1998)
6. Odersky, M., Zenger, M.: Independently extensible solutions to the expression problem. In: FOOL '05. (2005)
7. Buchlovsky, P., Thielecke, H.: A type-theoretic reconstruction of the visitor pattern. In: MFPS XXI. Electronic Notes in Theoretical Computer Science (ENTCS) (2005)
8. Oliveira, B.C.d.S., Wang, M., Gibbons, J.: The visitor pattern as a reusable, generic, type-safe component. In: OOPSLA '08. (2008)
9. Böhm, C., Berarducci, A.: Automatic synthesis of typed lambda-programs on term algebras. Theoretical Computer Science **39** (1985) 135–153
10. Parigot, M.: Recursive programming with proofs. Theor. Comput. Sci. **94**(2) (1992) 335–356
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
12. Girard, J.Y., Taylor, P., Lafont, Y.: Proofs and types. Cambridge University Press (1989)
13. Moors, A., Piessens, F., Odersky, M.: Generics of a higher kind. In: OOPSLA '08. (2008)
14. Bracha, G., Cook, W.: Mixin-based inheritance. In: OOPSLA '90, ACM Press (1990) 303–311
15. Schärli, N., Ducasse, S., Nierstrasz, O., Black, A.: Traits: Composable units of behavior. In: ECOOP '03. Volume 2743., Springer-Verlag (2003) 248–274
16. Bruce, K., Schuett, A., van Gent, R., Fiech, A.: Polytoil: A type-safe polymorphic object-oriented language. ACM Trans. Program. Lang. Syst. **25**(2) (2003) 225–290
17. Igarashi, A., Viroli, M.: Variant parametric types: A flexible subtyping scheme for generics. ACM Trans. Program. Lang. Syst. **28**(5) (2006) 795–847

18. Torgersen, M.: The expression problem revisited. In: ECOOP '04. (2004)
19. Clifton, C., Leavens, G.T., Chambers, C., Millstein, T.: MultiJava: Modular open classes and symmetric multiple dispatch for Java. In: OOPSLA '00. (2000) 130–145
20. Zenger, M., Odersky, M.: Extensible algebraic datatypes with defaults. In: ICFP '01. (2001) 241–252
21. Löh, A., Hinze, R.: Open data types and open functions. In: PPDP '06. (2006) 133–144
22. Poll, E.: Subtyping and inheritance for inductive types. In: Informal proceedings of the 1994 TYPES Workshop. (1997)
23. Lopez-Herrejon, R.E., Batory, D.S., Cook, W.R.: Evaluating support for features in advanced modularization technologies. In: ECOOP '05. (2005) 169–194
24. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
25. Cardelli, L.: Extensible records in a pure calculus of subtyping. In: Theoretical Aspects of Object-Oriented Programming, MIT Press (1994) 373–425
26. Cremet, V., Altherr, P.: Adding type constructor parameterization to Java. Journal of Object Technology **7**(5) (2008) 25–65
27. Odersky, M., al.: An overview of the Scala programming language (second edition). Technical Report IC/2006/001, EPFL Lausanne, Switzerland (2006)
28. Chambers, C., Leavens, G.T.: Typechecking and modules for multimethods. ACM Transactions on Programming Languages and Systems **17**(6) (1995) 805–843
29. Prehofer, C.: Feature-oriented programming: A fresh look at objects. In: ECOOP '97, Springer-Verlag (1997) 419–443
30. Krishnamurthi, S., Felleisen, M., Friedman, D.P.: Synthesizing object-oriented and functional design to promote re-use. In: ECOOP '98, Springer-Verlag (1998) 91–113
31. Palsberg, J., Jay, C.B.: The essence of the visitor pattern. In: Proc. 22nd IEEE Int. Computer Software and Applications Conf., COMPSAC. (19–21 1998) 9–15
32. Vlissides, J.: Pattern hatching - visitor in frameworks (1999)
33. Martin, R.C.: The Dependency Inversion Principle. The C++ Report (May 1996)
34. Ingalls, D.H.H.: A simple technique for handling multiple polymorphism. In: OOPSLA '86. (1986) 347–349
35. Ernst, M., Kaplan, C., Chambers, C.: Predicate dispatching: A unified theory of dispatch. In: ECOOP '98, London, UK, Springer-Verlag (1998) 186–211
36. Bruce, K.B.: Some challenging typing issues in object-oriented languages. Electr. Notes Theor. Comput. Sci. **82**(7) (2003)
37. Oliveira, B.C.d.S., Hinze, R., Löh, A.: Extensible and modular generics for the masses. In: TFP '06. (2006) 109–138
38. Oliveira, B.C.d.S.: Genericity, Extensibility and Type-Safety in the Visitor Pattern. PhD thesis, University of Oxford (2007)
39. Swierstra, W.: Data types à la carte. Journal of Functional Programming **18**(4) (2008) 423–436
40. Garrigue, J.: Code reuse through polymorphic variants. In: Workshop on Foundations of Software Engineering. (2000) 93–100
41. Ernst, E.: The expression problem, Scandinavian style. In Lahire, P., al., e., eds.: MASPEGHI 2004. (2004)
42. Nystrom, N., Chong, S., Myers, A.C.: Scalable extensibility via nested inheritance. In: OOPSLA '04, ACM Press (2004) 99–115
43. Nystrom, N., Qi, X., Myers, A.C.: J&: nested intersection for scalable software composition. In: OOPSLA '06, ACM (2006) 21–36
44. Qi, X., Myers, A.C.: Sharing classes between families. In: PLDI '09. (June 2009)