EPiC Series in Computing

Volume 45, 2017, Pages 16–28

SCSS 2017. The 8th International Symposium on Symbolic Computation in Software Science 2017

Integration of a Decentralised Pattern Matching
Venue for a New Paradigm Intermarriage*

Seyed Hossein Haeri[1] and Sibylle Schupp[2]

[1] Université catholique de Louvain, Belgium
hossein.haeri@ucl.ac.be
[2] Hamburg University of Technology, Germany
schupp@tu-harburg.de

Abstract

We provide a new technique for pattern matching that is based on components for each match. The set of match statements and their order is open for configuration at the right time and takes place in a feature-oriented fashion. This gives rise to a solution to the Expression Problem in presence of defaults. It takes a lightweight discipline to develop components for our technique. Their use for configuration of the pattern match, however, is virtually automatic.

1 Introduction

Expression Problem (EP) [5, 32, 39] is a recurrent problem in the field of Programming Languages, for which a wide range of solutions have thus far been proposed: [38, 22, 35, 25, 1, 40], to name a few. EP is the challenge of finding an implementation for an algebraic datatype (ADT) – defined by its cases and the functions on it – that:

E1. is *bidirectionally extensible*, i.e., both new cases and functions can be added.

E2. provides *weak static type safety*, i.e., applying a function f on a statically[1] constructed ADT term t should fail to compile when f does not cover all the cases in t.

E3. upon extension, forces *no manipulation or duplication* to the existing code.

E4. accommodates *separate compilation*, i.e., compiling the extension imposes no requirement for repeating compilation or type checking of existing code. Such static checks should not be deferred to the link or run time either.

*partially funded by the German Research Council (DFG) and partially by the SyncFree project in the European Seventh Framework Programme under Grant Agreement 609551

[1] If the guarantee was for dynamically constructed terms, we would have called it strong static type safety.

EP has often been understood with focus on ADT extension. The key to our EP solution (in presence of defaults [41]), however, is that, even when ADTs do not extend one another, they sometimes share cases:

$$\alpha_1 ::= \mathit{Num}(\mathsf{int}) \mid \mathit{Add}(\alpha_1, \alpha_1) \mid \mathit{Sub}(\alpha_1, \alpha_1) \qquad \alpha_2 ::= \mathit{Num}(\mathsf{int}) \mid \mathit{Add}(\alpha_2, \alpha_2) \mid \mathit{Neg}(\alpha_2).$$

We promote ADT cases (i.e., *Num*, *Add*, *Sub*, and *Neg* in this case) into their own independent but ADT-parameterised presentations. Our solution builds on the use of components for the ADT cases as well as functions defined on each case. Components, in our model [12, §5][16], divide the duty between component vendors and component users (a.k.a. ADT implementer), for both of whom they demand coding discipline. Our technique demands coding discipline. Yet, whilst the weight of that is comparable for the component vendor to our former solution given in [15], the component user's job is now virtually automatic.

Our technique works by *integration of a decentralised pattern matching*: Instead of centralising the pattern matching in a single place, we distribute it amongst components that correspond to the ADT cases. With its excessive reliance on components, our technique is an instance of Component-Based Software Engineering (CBSE) [34, §17],[29, §10]. The way components need to be employed by the component user is, on the other hand, an instance of Feature-Oriented Programming (FOP) [28]. Hence, an intermarriage between the two paradigms.

Here is the list of our contributions:

1. We apply our technique to yet another EP generalisation called Expression Families Problem (EFP) of Oliveira [24]. We do that for the Equality Test exercise (Section 3) and a significantly generalised variation of the Narrowing exercise (Section 4) both of which designed by Oliveira. Our presentation in this paper is using the standard EP showcase: Integer Arithmetic Expressions, our Scala codebase for which being available online.

2. We explain how our technique solves EP (Section 3) and how it mixes ideas from both CBSE and FOP (Section 3).

3. We compare our technique with a closely-related one (Section 4), discuss its generality (Section 4), and report another application where the same technique was used for a set of six lazy languages. Details of the latter usecase can be found at [12, §9].

A selection of the rich literature on EP and EFP is discussed in Section 5. Conclusion and future work also come in Section 5. This paper assumes basic Scala.

2 Tools and Notations

In this section, we provide a short explanation of our basic components and how to mix them. The purpose is to lay foundations for our later developments.

We begin by recalling that, to accommodate ADT extension, Scala employs the familiar OOP inheritance to give extensible ADTs a first-class support. Hence, a core ADT and the code written in terms of it can be reused by extensions [23]. The Scala implementation of our technique takes that facility on-board. It also makes ADT cases stand-alone entities – forming a fully fledged approach, manifested in [13, 12]. In the accompanying formalism [15]: (i) an ADT case is represented using a *component* and is ranged over by γs; (ii) the ADT representation forms a *family* that simply combines components. Families are ranged over by Φs:

$$\mathsf{family}\ \ \Phi_1 = \mathit{Num} \oplus \mathit{Add} \oplus \mathit{Sub} \quad //\text{for } \alpha_1 \qquad \mathsf{family}\ \ \Phi_2 = \mathit{Num} \oplus \mathit{Add} \oplus \mathit{Neg} \quad //\text{for } \alpha_2$$

Whilst the resulting benefits are beyond this paper, here, we do make use of both the (implementation) approach and the formalism. The introduction of Φ_1 and Φ_2 above are examples of the latter. (Here, *Num*, *Add*, *Sub*, and *Neg* are components with known constructor signatures.) We start by an illustration of the corresponding component of *Add* in Scala

```
class Add[E <: IAE[E],
          A <: Add[E, A] with E](left: E, right: E) {/* Add's Behaviour */}
```

in which the type constraints (i.e., what comes between the square brackets) play a key role. They state that `Add` can only be used by an ADT `E` that can present `A` as a witness for its `Add` case. (`IAE` is the root of our type hierarchy of *I*nteger *A*rithmetic *E*xpressions. And, we name `E` after *E*xpression.) Using the formalism, one can state the same restriction via the so-called 'requires' interface [34, 29] of the corresponding component: component *Add*<$F \lhd$ *Add*> {...}. The 'requires' interface above is the "$F \lhd$ *Add*" portion, which states the following: The family to be substituted for the *family parameter* F needs to at least have *Add* in its component combination. (Both Φ_1 and Φ_2 are eligible, for instance.) Inside the body of the *Add* component, one can refer to the properties of such a family via the parameter F. Section 4 provides a discussion on that basis.

Here is how to employ `Add` (and other components) to implement Φ_1.

```
1 trait Phi1 extends IAE[Phi1]
2 case class Num1(n: Int)                  extends Num[Phi1, Num1](n) with Phi1
3 case class Add1(left: Phi1, right: Phi1) extends
4   Add[Phi1, Add1](left, right) with Phi1
5 case class Sub1(left: Phi1, right: Phi1) extends
6   Sub[Phi1, Num1, Sub1](left, right) with Phi1
```

Observe how, in line 4 above, `Add1` substitutes `Phi1` and `Add1` for `E` and `A` in `Add`, respectively. In a similar fashion, one can combine `Num`, `Add`, and `Neg` to get `Phi2` (for α_2). We would like to remind that `Num1`, `Add1`, and `Sub1` need not to implement any *Num*, *Add*, and *Sub* **behaviour**. All their required behaviours are already implemented by the respective off-the-shelf components: `Num`, `Add`, and `Sub`. (For example, although not detailed here, the behaviour of `Add` is what comes between its curly braces.) The cases of `Phi1` get their desirable behaviours for free by inheritance (lines 2, 4, and 6 above). Provision of components such as `Add` and `Neg` is the role envisioned for the component vendor. It is their use (lines 2 to 6 above) that is the role of the component user. (See Section 3 for more.)

Finally, for a given ADT α, Haeri [12, §6.1] introduces the notion of a *compatible extension* α', denoted by $\alpha' <_{\mathscr{C}} \alpha$. We use the same notation. It suffices for the reader to know that α', in such a case: extends α; does not remove any of α's cases; and, does not replace α's cases by "incompatible" ones (in a sense that we do not detail here).

3 Equality

The purpose of the Equality Test is to provide a statically safe solution for *multiple dispatching* that is also extensible. The driving example is structural equality between expressions $Num(n) = Num(n')$ iff $n = n'$ (1); $Add(e_1, e_2) = Add(e'_1, e'_2)$ iff $e_1 = e'_1 \wedge e_2 = e'_2$ (2); $Sub(e_1, e_2) = Sub(e'_1, e'_2)$ iff $e_1 = e'_1 \wedge e_2 = e'_2$ (3); and, $Neg(e) = Neg(e')$ iff $e = e'$ (4). Like Oliveira, we offer a solution that simulates multi-methods [3, 4] in Scala.

We embark on the exercise by offering a new set of components that build on top of the Scala introduced in Section 2. Call these new components the *equality component*s. Each equality component takes the exclusive responsibility of its own part of structural equality test. (That

is one and only one of the Equations 1 to 4.) This is done using a method called `this_case`. For example, `EqN` and `EqA` below handle Equations 1 and 2, respectively:

```
1  trait EqN[E <: IAE[E]] extends EqB[E] {//More on EqB soon.
2    override def this_case: (E, E) => Boolean = {
3      case (n1: Num[_, _], n2: Num[_, _]) if n1.n == n2.n => true
4      case (e1, e2) => super.this_case(e1, e2)
5    }
6  }
7  trait EqA[E <: IAE[E]] extends EqB[E] {
8    override def this_case: (E, E) => Boolean = {
9      case (a1: Add[_, _], a2: Add[_, _]) =>
10       areeq(a1.left, a2.left) && areeq(a1.right, a2.right)
11     case (e1, e2) => super.this_case(e1, e2)
12   }
13 }
```

Note that line 3 above is the only non-trivial equality case for `EqN`. Likewise is line 10 for `EqA`. Hence, lines 4 and 11 forward the other cases of equality test to the **next level above** in the stack of mixed-in equality components. (See `eqer` below.) In a similar fashion, one gets `EqS` and `EqG` for `Sub` and `Neg`, respectively.

We now give an example on how such components mix together to solve the problem: The mixed traits in `eqer` below

```
1  object eqer extends
2    EqB[Phi1] with EqN[Phi1] with EqA[Phi1] with EqS[Phi1] with EqF[Phi1]
```

enable us to perform the equality tests like the one below. For α_1, the following two expressions are checked for being the same: "3+5" (`tp_five1`) and "3+5−1" (`tpfm_one1`), and, actually get the true negative from `println("tp_five1 == tpfm_one1?" + eqer.areeq(tp_five1, tpfm_one1))`, where `val tp_five1 = Add1(Num1(3), Num1(5))` (i.e., $3 + 5$ for α_1) and `val tpfm_one1 = Sub1(Add1(Num1(3), Num1(5)), Num1(1))` (i.e., $3 + 5 - 1$ for α_1). (More on `areeq` shortly.) Similar checks are possible likewise for α_2 via the following component combination:

```
EqB[Phi2] with EqN[Phi2] with EqA[Phi2] with EqG[Phi2] with EqF[Phi2]
```

The last two pieces of puzzle are `EqB` and `EqF`. (C.f. line 2 of `eqer`).

```
1  trait EqB[E <: IAE[E]] {
2    def this_case: (E, E) => Boolean = {case _ => false}
3    def areeq(e1: E, e2: E): Boolean
4  }
5  trait EqF[E <: IAE[E]] extends EqB[E] {
6    override def areeq(e1: E, e2: E): Boolean = super.this_case(e1, e2)
7  }
```

`EqB` is the base class of all the equality components. (That is, the trailing `B` is for "base".) The role of `EqB` is two-fold: its `this_case` method implements the default equality test case (to false); and, it outlaws the use of equality components until a concrete `areeq` is mixed in. (Note that the method `areeq` in line 3 of `EqB` is abstract.) Finally, a concrete implementation of `areeq` is provided by `EqF`, which simply forwards the overall task to the next level above in the mixin stack (line 6).

With this wiring, a call like `areeq(tp_five1, tpfm_one1)` will start the equality test from `EqF`; go up to `EqS`; and, keep climbing the ladder up until it finds the right handler, if any. In the mean time, if it needs to test the equality of sub-expressions, `areeq` (which integrates the distributed pattern matching) will be called. An example of the latter calls is line 10 of `EqA`.

Our solution caters for extensibility by leaving open the possibility of mixing-in new equality components without the need to touch the existing equality cases. Note that, in this very

instance, the structural exercise happened to come with a default case. As noticed first by Zenger and Odersky [41] and several others afterwards, a default is not necessarily available. Section 4 contains an example where our solution works in the absence of default cases.

Expression Problem Let us take our time here to discuss how the technique addresses EP. We proceed by a discussion on each EP concern except E4 that is like E3: (See Section 1.)

E1. Addition of both new cases and functions is possible: Adding new cases takes implementation of new components like those presented in Section 2. Adding new functions amounts to implementing the respective components (like the equality components in this section) for the distributed pattern matching and integrating them.

E2. It turns out that the support for this concern is a matter of which host language it is used in. For example, in C++ where metaprogramming is commonly employed to get the compiler (automatically) try to pattern match, non-exhaustive functions will be rejected at compile-time. In other languages like Scala that we use for presentation here and that perform pattern matching at runtime, such a support is not available. See the last paragraph of Section 4 for more.

E3. Addition of new components or new integrations has no impact on the existing code. The programmer may very well, upon the addition of new cases or functions, decide to refactor the existing code say for new needs are revealed accordingly. That, however, will not be a consequence of the addition itself.

Feature-Oriented Programming We would like now to redraw the reader's attention to the integration part our technique (i.e., integration of a decentralised pattern matching). An alternative interpretation of that action reveals the connection with feature-oriented programming: Upon the integration, each component also acts like a feature in that it is an increment in functionality. One can consider `EqB` as a basic service to which each component adds feature upon the integration. Take `eqer` for example, where each of `EqN`, `EqA`, and `EqS` is an increment in the functionality forming the total service required for α_1.

One might at this point be confused about why we argue that this is a usage of a new paradigm. After all, in CBSE, a system is developed by **assembl**ing together the preexisting components [27, §5.3]. Such a confusion is rooted in the similarity between CBSE and FOP. Of course, like also pointed out by Mernik [21], what is common between the two paradigms is that they both get the end-user involved in the software development. Yet, the former breaks a problem into the so-called components, which are to function so long as their 'requires' are provided. The latter, on the other hand, enables variation by offering a basic service on top of which other features can be added. That is, unlike features, components are not designed to add to on top of a basic service.

Roles The equality components here are to be provided by the component vendor. A quick comparison with Section 2 reveals it that the provision of such components takes comparable effort with that required by components like `Add`. The component user's role, on the other hand, is simply mixing the equality components to get `eqer` and the like and enjoy the ready-made services without any further effort. That is why we say that the effort expected from the component user is virtually zero when using this technique. (Consult Section 4 for more.)

Technicality In fact, in our codebase, line 10 of `EqA` is not exactly as shown here. Four type casts are required that we drop here for presentational reasons: For example, `a1.left` needs to be `a1.left.asInstanceOf[E]`. These casts are not intrinsic to our solution; they are a consequence of Scala's choice for type erasure in pattern matching. (It is not exclusively our solution that is challenged by Scala's type erasure. Madsen and Ernst [19, §4] report similar issues in their more recent work on Virtual Classes [6, 7].) In other words, in an unerased host language, we could simply write `case (a1: Add[E, _], a2: Add[E, _])` in line 9 of `EqA` and discard the cast. It is also worth noting that these casts are guaranteed not to fail at runtime; `a1` and `a2` are already both of type `E`. Recall from Section 2 that the first type parameter of `Add` ensures that `left` and `right` fields are of type `E`.

4 Conversion and Narrowing

Following Oliveira [24], we say an expression is **narrow**ed when all its ADT cases of a given group are cancelled into other case combinations that are deemed to be equivalent. It is common in the Programming Languages community to provide extensions to a core language such that the extension programs would then be narrowed to the core (for evaluation and the like). For example, GPH and Utrecht Haskell are both developed like that. Oliveira shows how his Modular Visitor Components (MVCs) can be leveraged in favour of correctness for narrowing as a static guarantee that the result of this process will not contain instances of the unwanted ADT cases; it will instead contain other case combinations that are deemed equivalent.

In this section, we generalise the narrowing exercise to conversion from one ADT to another. Being based on a single ADT, narrowing is a single dispatching problem. As will be discussed later, however, with the level of correctness that it guarantees, our solution for the conversion exercise also addresses type-safe multiple dispatching – with minor reservations in the absence of a default case. (Note that there is no default case in the conversion exercise.)

We discuss expression conversion from α to α' such that $\alpha <_{\mathscr{C}} \alpha_1$ and $\alpha' <_{\mathscr{C}} \alpha_2$. (Recall from Section 2 that such a conversion is from any compatible extension to α_1 to equivalent compatible extension to α_2.) Note that both α_1 and α_2 contain *Num* and *Add*. Thus, the conversion only needs to rewrite *Sub* expressions of α into a combination of α' cases. (See Equation 7.) Hence, applying this conversion from an ADT $\alpha_3 ::= \mathit{Num}(\mathsf{int}) \mid \mathit{Add}(\alpha_3, \alpha_3) \mid \mathit{Sub}(\alpha_3, \alpha_3) \mid \mathit{Neg}(\alpha_3)$ to itself – which is a compatible extension to both α_1 and α_2 – will result in narrowing for α_3. (Because the conversion will cancel a *Sub* expression of the α_3 into the α_3 equivalent.) Supposing the implementation `Phi3` for α_3 using the developments of Section 2, one can achieve the desirable narrowing using `narr` below. (More on `NSub` later on.)

```
object narr extends NSub[Phi3, Num3, Add3, Sub3, Neg3]
```

Given the `val tpfmo3 = Sub3(Add3(Num3(3), Num3(5)), Num3(1))` transliteration of "3+5−1" in α_3, the expression `narr.convert(tpfmo3)` will return "3+5+(−(1))" – again, transliterated in α_3. (More on the `convert` method soon.)

Like our solution to the equality exercise, our solution here too is based on integration of a decentralised pattern matching. We do so using yet another group of components: *conversion components* (to be provided by the component vendor). Our conversion components are like equality components of Section 3 except that they concern conversion as opposed to equality. However, as we will see, conversion components are more restrictive in their type parameters. They provide stronger static safety in a sense that will be discussed in Section 4: $[\![\mathit{Num}(n)]\!] = \mathit{Num}(n)$ (5); $[\![\mathit{Add}(e_1, e_2)]\!] = \mathit{Add}([\![e_1]\!], [\![e_2]\!])$ (6);

$[\![Sub(e_1, e_2)]\!] = Add([\![e_1]\!], Neg([\![e_2]\!]))$ (7); and, $[\![Neg(e)]\!] = Neg([\![e]\!])$ (8).
The equations above show the cases of the conversion function $[\![.]\!]$. The only non-trivial case is Equation 7. Similarly, the only non-trivial conversion component is `CS2AN`, which is responsible for Equation 7. `CS2AN` differs from our equality components in a number of ways.

```
1  trait CS2AN[
2    E1 <: IAE[E1], N1 <: Num[E1, N1] with E1, S1 <: Sub[E1, N1, S1] with E1,
3    E2 <: IAE[E2], N2 <: Num[E2, N2] with E2, A2 <: Add[E2, A2]     with E2,
4    G2 <: Neg[E2, N2, G2]            with E2] extends CB[E1, E2] {
5      abstract override def this_case(e1: E1): E2 = e1 match {
6        case raw_s1: Sub[_, _, _] => {
7          val s1 = raw_s1.cast[Sub[E1, N1, S1]].get
8          new Add[E2, A2](convert(s1.left), new Neg[E2, N2, G2](convert(s1.right)))
9        }
10       case _ => super.this_case(e1)
11   }
12 }
```

Firstly, it is parameterised over both the source and the destination identity types (`E1` and `E2` in lines 2 and 3, respectively). This is the multiple dispatching nature of conversion we formerly spoke about. Secondly, it demands the availability of certain cases for the ADTs: line 2 demands `N1` and `S1` for `E1` and lines 3 to 4 demand `N2`, `A2`, and `G2` for `E2`. Using the formalism briefed about in Section 2, one describes that as
component $CS2AN\!<\!F_1 \lhd Num \oplus Sub, F_2 \lhd Num \oplus Add \oplus Neg\!>\ \{\cdots\}$.

Each conversion component places type constraints that solely mirror the cases that are relevant to their own part of the conversion recipe (namely, Equations 5 to 8). This is what we call *minimal shape exposure*. See Section 4 the consequences on dictating the type safety.

The third difference is that `CS2AN` is uneven regarding the cases it demands from the two ADTs. Amongst our conversion components, this latter difference is unique to `CS2AN`. The only place where one case gets replaced by a combination of **other**s is, after all, Equation 7.

To complete the picture, we would point out that: The portion of `CS2AN` that is actually responsible for the right-hand-side of Equation 7 is lines 7–8. Note that the cast in line 7 is again because, due to erasure, Scala loses the relevant type information upon the pattern matching in line 6. Yet, just like the case for `EqA` in Section 3, the cast in line 7 is guaranteed to never fail. The difference here is that, instead of Scala's built-in conversion mechanism, we are this time employing the Shapeless `cast` method in line 6.

Mixing the components into `NSub` – which is on the component user – is similar to `eqer` in Section 3, except that it is multistaged:

```
1  trait CSub[... /* revisited later */ ...] extends CB[E1, E2] with
2    CN2N[E1, N1, E2, N2]              with //Equations 5
3    CA2A[E1, A1, E2, A2]              with //Equations 6
4    CG2G[E1, N1, G1, E2, N2, G2]      with //Equations 8
5    CS2AN[E1, N1, S1, E2, N2, A2, G2] with //Equations 7
6    CF[E1, N1, E2, N2]
7  trait NSub[E <: ..., N <: ..., A <: ..., S <: ..., G <: ...] extends
8    CSub[E, N, A, S, G, E, N, A, G]
```

Instead of being specific to a conversion from α_1 to α_2, `CSub` combines the respective components for the generalised conversion promised earlier on. `NSub`, then, unifies the source and destination of the conversion to perform narrowing for $\alpha <_{\mathscr{C}} \alpha_3$. Note that, because of the above multi-staging, one can also use `CSub` itself directly for the conversion:

```
object converter extends CSub[Phi3, Num3, Add3, Sub3, Neg3, Phi2, Num2, Add2, Neg2]
```

So that the call `{converter.convert(tpfmo3)` would return `Add2(Add2(Num2(3), Num2(5)), Neg2(Num2(1)))` which is essentially "$3 + 5 + (-(1))$" transliterated in α_2.

Now that we are done with how to combine the components, we move to explaining the other conversion components than `CS2AN`. We start from the trait `CB` (for *C*onversion *B*ase) used in line 4 of `CS2AN`. This trait is the counterpart of `EqB` in Section 3.

```
1  trait CB[E1 <: IAE[E1], E2 <: IAE[E2]] {
2    def this_case(e1: E1): E2
3    def convert(e1: E1): E2}
```

However, given that there is no default case in the conversion exercise, instead of inheriting from `CB`, the conversion counterpart of `EqF` inherits from `CN2N`.

```
1  trait CF[E1 <: IAE[E1], N1 <: Num[E1, N1] with E1,
2           E2 <: IAE[E2], N2 <: Num[E2, N2] with E2] extends CN2N[E1, N1, E2, N2] {
3      override def convert(e1: E1): E2 = super.this_case(e1)
4  }
```

`CN2N` is the corresponding conversion component of Equation 5 (i.e., from *Num* to *Num*). `CN2N` is the component for the induction's base case. (Numbers are converted to themselves, hence, no recursion. Notice that, in Equation 5, there is no recursive $[\![.]\!]$ call.) Likewise, the method `this_case` of `CN2N` (line 2 below) is not `abstract`:

```
1  trait CN2N[...] extends CB[E1, E2] {
2    override def this_case(e1: E1): E2 = e1 match {
3      case n1: Num[_, _] => new Num[E2, N2](n1.n)
4      case _             => throw new ...
5    }
6  }
```

Type Heaviness The type treatment left out by ellipsis at line 1 of `CSub` performs F-Bounding for `E1`, `N1`, `A1`, `S1`, `G1`, `E2`, `N2`, `A2`, and `G2`. That, admittedly, is a high volume for boilerplate code. One may wonder how, then, it is that we consider the component user's job automatic with such a volume for toy examples. Firstly, if one is indeed to always key in all such type treatments, simple copy/paste does. And, the standard services of the compiler are of help: usage of missing type parameters (i.e., ADTs or their cases) will be outlawed statically. (See also the upcoming discussion on technicality.) Secondly, different languages have different mechanisms to reduce repetition in presence of high-volume type treatments. For example, C++ offers alias templates and Scala offers virtual types. (Due to space restrictions, we cannot demonstrate either in this paper.) Thirdly, automatic generation of such F-Boundings (say, using macro expansion or similar offline tools) is an easy task.

Technicality Whilst we are done with the outline, in this section, we provide extra insight into our technique. We, then, report an additional development on the narrowing exercise.

The astute reader is likely to have already noticed it that the method `this_case` of `CB` is supposed to return an `E2`. Yet, strictly speaking, none of the `this_case` implementations shown here indeed do so. For example, in the case of `CS2AN` below, the method returns an `Add` instance in line 8. That is possible because an implicit conversion is, in fact, also assumed. Here is a more realistic signature of `this_case` method of `CB`: `def this_case(e1: E1)(implicit incarnator: ICBase => E2): E2`, where `incarnator` is what implicitly converts say an `Add` to an `E2`. (`ICBase` is the root of our ADT case components.) Whilst we dropped the presentation of that in Section 2 for brevity, in reality, each family implementation is required to provide an `incarnator` for its own cases. See [12, §8.3] for more.

On a different note, we would now present the promised static safety comparison between our equality components and our conversion components. Mixing-in an equality component for an ADT α lacking the respective case γ will not fail at runtime – the corresponding pattern match of γ will simply never succeed; no compile error will be emitted either. On the contrary, mixing a conversion component for such an α and γ will be rejected statically. Would the programmer mistakenly attempt to access other cases of either ADT than the explicitly nominated ones (e.g., in lines 2 to 4 of `CS2AN`, compilation will fail.

This is because the equality components all only take one type parameter: that of the ADT they are being used for. In other words, they do not put any extra restrictions on the ADT. In particular, the compiler is provided with no information on cases that the equality component expects from the ADT. Contrast that with conversion components (e.g., `CS2AN`) that explicitly state all their expected ADT cases (e.g., lines 2 and 4). Yet, in its body, an equality component addresses its part of the equality testing by assuming the respective case for the ADT. (See lines 9 and 10 of `EqA` for instance.) Similar unannounced assumptions can become unsafe by say constructing a γ instance for the use of α's `incarnator`. The `incarnator` will, however, fail at runtime in such a situation because there is no way to convert a γ to an α.

To prevent all that, in his formalism, Haeri devises a rule called (WF-VCase) for the static semantics that outlaws attempts for accessing unrequested cases [12, Figure 6.9]. In fact, whilst they might initially look admissible as they are, the equality components are designed to be statically rejected by (WF-VCase). The way this is achieved becomes more clear by noticing the following: A mistaken attempt to employ the formalism for an equality component such as `EqA` may look like $\mathsf{component}\ \mathit{EqA}\langle F \lhd \epsilon\rangle \{\ldots \mathit{this\_case}(\mathit{F.Add}\ a_1, \mathit{F.Add}\ a_2)\ \{\ldots\}\ldots\}$ (where ϵ represents an empty list). But, then, for every γ, trying to access $F.\gamma$ in the body of *EqA* will (correctly) fail. ($F.\gamma$ here is the case γ of the ADT substituted for F in *EqA*.) On the contrary, accesses like $F_1.\mathit{Num}$ in the body of *CS2AN* will succeed for the respective case is already in the 'requires' interface of *CS2AN*.

Having said that about the formalism, we need to stipulate that the Scala implementation of the technique presented in this paper is not as strongly safe as the formalism. More specifically, the technique relaxes exhaustiveness in the absence of a default (say as in Section 4): If the component user fails to mix all the required components that correspond to the cases of a given ADT instance, the pattern match will run out of options, causing a runtime error due to its failure to serve the instance. It turns out that the lack of exhaustiveness in the absence of a default is not considered totally unacceptable. For example, EP solutions like LMS [33], MVCs [24], and Torgersen's second solution [38] all have the same issue.

Comparison There is a categorical difference between the technique presented in this paper and that of Haeri and Schupp [13]. Unlike the latter, the former addresses function concerns of the Expression Compatibility Problem [15] using a verified software product-line [36, 37]. (In fact, the former is the only instance of a Feature-Family-Product-Based Analysis [37] that we are aware of.) The difference becomes further clear when one considers the coding discipline each group of components is shipped to their client with. For example, compare the following two: how components are used to get `eqer` in Section 3 versus how they are used to get `OpSem` in [13, Fig. 5]. In order to use the latter, the client has to provide a function (say called `eval`) that manually distributes the task amongst the (semantics) components by explicitly calling the appropriate ones after a client-side pattern match. For the former, the client assembles the appropriate components using simple mixin composition; and, uses a readily available method of the mixin stack that automatically performs the pattern matching. In the terminology of Sommerville, the former is a *sequential composition* [34, §17.3] whilst the latter is an *additive*

one. Nevertheless, ensuring soundness of composition is similar between the two techniques. Both techniques, after all, build on top of case components (Section 2).

Generality It is worth noting that the resort to stackability of mixins is not fundamental to the solutions presented in this paper. In a host language like C++ where template specialisation is provided, the decentralisation components all have the same name, and, the compiler integrates the pattern matching automatically. In the absence of that, we simulate it by manually directing the flow of execution.

Other Applications Using similar techniques to those in this section, one can gain narrowing for functions defined on an ADT (rather than just its cases). Combining the result with the compatible extension techniques developed in [12, §8.4.1] gives rise to a manual simulation for the bidirectional adaptation of $J\&_s$ [30]. That is, one uses the latter techniques to get from an ADT to its extension; the former techniques can get one back to a core ADT.

On the other hand, using the technique of this paper, we completed both the equality and the (generalised) narrowing exercises in the laziness world as well. Besides, we took the second exercise one more step ahead by also extending it to heaps and derivation trees (in its obvious pointwise manner). Armed with those, a phrase like `s3opsem.eval(convert(g), convert(e)) == convert(s2opsem.eval(g, e))` can be used to test whether, for a given expression e and a heap Γ, the following commutativity property holds: $\Gamma : e \Downarrow_{\mathcal{S}_2} \Delta : v$ iff $[\![\Gamma]\!] : [\![e]\!] \Downarrow_{\mathcal{S}_3} [\![\Delta]\!] : [\![v]\!]$. In words, the equation says: 'One can either perform an $\mathcal{S}_2$ [11] evaluation and then convert to $\mathcal{S}_3$ [14], or convert first and perform the evaluation under $\mathcal{S}_3$— the result will be the same.' That is, conversion and evaluation commute from $\mathcal{S}_2$ to $\mathcal{S}_3$. (C.f. [12, §9.2.2] for more.)

5 Concluding Remarks

Related Work Oliveira was the first to define EFP and discuss the necessity of solving it. He also offered MVCs [24] as an EFP solution. Oliveira and Cook [25] back this work up by the powerful and simple concept of object algebras [10]. Object algebras outperform MVCs in that they do not require the clunky wiring of the VISITOR pattern [8]. Later, Oliveira et al. [26] address some awkwardness issues faced in their former paper upon composition of object algebras. Rendel, Brachthäuser and Ostermann [31] add ideas from attribute grammars to the latter work to get reusable tree traversals.

As also pointed out by Black [2], an often neglected factor about solutions to EP is the complexity of term creation. That complexity increases from one work to the next in the above literature on EFP. The symptom develops to the extent that it takes Rendel, Brachthäuser and Ostermann 12 non-trivial lines of code to creat a term representing "3 + 5". Of course, those 12 lines are not for the latter task exclusively and enable far more reuse. Yet, that is so heavyweight and the latter work uses automatic code generation for term creation.

Our understanding is that, in EFP, the term 'component' is used as a correspondent for ADT cases. This guess is backed up at various occasions in his seminal paper, where Oliveira discusses his EFP solutions.[2] We admit that the notion of components can be hard to agree upon. Yet, we have to also express our failure in pinpointing the entity in the two latter works of Oliveira et al. that is to correspond with the above understanding of the term 'component'. The obvious first guess – that goes, for example, with the definition of Herzum and Sims from

[2]For instance, in [24, Fig. 8], he names his solution for the Equality Test exercise `ExtendedComponents` rather than `ExtendedComponent`. Note the missing plural 's' at the end of the second name.

components [17] – is their object algebras. However, to us, their object algebras are rather omnipotent packs of components (ADT cases). We are not sure how to make a meaningful comparison between components and packs of components.

Our final remark about MVCs is that, rather than components in their CBSE sense, MVCs are components in a Component-Oriented Programming [20] sense. Testimony to that is the excess of inside knowledge about MVCs that is crucial for managing the Equality Test [24, Fig. 8]. This is in contrast with CBSE components, where, for example, relying on the implementation details of **how** a component realises its interfaces is not acceptable. In CBSE, components are identified by their 'requires'/'provides' **interface**s.

Of the rich literature on EP we only consider a few that we deem close enough to this paper. Garrigue [9] solves EP using global case definitions that, at their point of definition, become available to every ADT defined afterwards. Per se, a function that pattern matches on a group of these global cases can serve any ADT containing the selected group. With OCaml's built-in support for Polymorphic Variants, this work is considerably simpler than that of ours. Yet, in contrast to this paper, Garrigue's work has no notion of components. Rompf and Odersky [33] employ a fruitful combination of the Scala features to present a very simple yet effective solution to EP using Lightweight Modular Staging (LMS). The support of LMS for E2 can be broken using an incomplete pattern matching. Yet, given that pattern matching is dynamic, whether LMS really relaxes E2 is debatable.

Out of the available EP solutions, we use the material in [23] with similarities to LMS [33], MVCs [24], and Polymorphic Variants [9].[3] We believe many EP solutions can be augmented by giving ADT cases support that is unbound to a particular ADT. We choose to build on top of LMS for its relative elegance and brevity. We choose to emulate Polymorphic Variants (in Scala) because that was the EP solution ADT cases are given the greatest freedom in.[4] Nonetheless, we minimise the drawbacks of ADT cases being global by promoting them to components. For a discussion on such drawbacks, see Black [2]. Although not quite related in the techniques used for solving EP, the recent work of Wang and Oliveira [40] is worth noting as well. This latter work is outstanding in its ease of use and the little number of advanced features it expects from the host language. Their approach uses covariant type refinement of return types. We, on the other hand, use type parameterisation and multiple-inheritance.

Conclusion and Future Work In this paper, we present a new technique for pattern matching organisation. By distributing the match statements amongst components, the matching remains fully flexible for configuration via component combination. Our technique mixes ideas from CBSE and FOP. The important result is a new solution to an old problem: EP (in presence of defaults). We show how our technique solves Expression Families Problem as well. Our immediate future work is adding support for exhaustiveness and addressing EP even in the absence of defaults. We need to also examine how well this technique scales for larger applications. Trying the technique in other host languages will form another open realm for research.

References

[1] P. Bahr and T. Hvitved. Parametric Compositional Data Types. In J. Chapman and P. B. Levy, editors, 4^{th} *MSFP*, volume 76 of *EPTCS*, pages 3–24, February 2012.

[3] Unlike LMS, we do not target Polymorphic Embedding [18] for DSLs. As such, we do not employ higher kinded types as they do. However, like LMS, we do not restrict the choice of components to predefined ones.

[4] Although the pivotal independence of ADT cases is not acknowledged by Garrigue [9].

[2] A. P. Black. The Expression Problem, Gracefully. In M. Sakkinen, editor, *MASPEGHI@ECOOP 2015*, pages 1–7. ACM, July 2015.

[3] C. Chambers and G. T. Leavens. Typechecking and Modules for Multimethods. *TOPLAS*, 17(6):805–843, 1995.

[4] C. Clifton, G. T. Leavens, C. Chambers, and T. D. Millstein. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In 15^{th} *OOPSLA*, pages 130–145, Minneapolis, Minnesota, USA, 2000. ACM.

[5] W. R. Cook. Object-Oriented Programming Versus Abstract Data Types. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *FOOL*, volume 489 of *LNCS*, pages 151–178, Noordwijkerhout (Holland), June 1990.

[6] E. Ernst. Higher-Order Hierarchies. In L. Cardelli, editor, 17^{th} *ECOOP*, volume 2743 of *LNCS*, pages 303–328. Springer, July 2003.

[7] E. Ernst. Reconciling Virtual Classes with Genericity. In D. E. Lightfoot and C. A. Szyperski, editors, 7^{th} *Joint Conf. Modular Prog. Lang.*, volume 4228 of *LNCS*, pages 57–72. Springer, 2006.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. AW Professional, October 1994.

[9] J. Garrigue. Code Reuse through Polymorphic Variants. In *FSE*, number 25, pages 93–100, 2000.

[10] J. V. Guttag and J. J. Horning. The Algebraic Specification of Abstract Data Types. *Acta Informatica*, 10:27–52, 1978.

[11] S. H. Haeri. Observational Equivalence and a New Operational Semantics for Lazy Evaluation with Selective Strictness. In Z. Majkic, S.-Y. Hsieh, J. Ma, I. M. M. El Emary, and K. S. Husain, editors, *TMFCS*, pages 143–150. ISRST, July 2010.

[12] S. H. Haeri. *Component-Based Mechanisation of Programming Languages in Embedded Settings*. PhD thesis, STS, TUHH, Germany, December 2014.

[13] S. H. Haeri and S. Schupp. Reusable Components for Lightweight Mechanisation of Programming Languages. In W. Binder, E. Bodden, and W. Löwe, editors, 12^{th} *SC*, volume 8088 of *LNCS*, pages 1–16. Springer, June 2013.

[14] S. H. Haeri and S. Schupp. Distributed Lazy Evaluation: A Big-Step Mechanised Semantics. In 22^{nd} *PDP*, PDP '14, pages 751–755, Washington, DC, USA, February 2014. IEEE.

[15] S. H. Haeri and S. Schupp. Expression Compatibility Problem. In J. H. Davenport and F. Ghourabi, editors, 7^{th} *SCSS*, volume 39 of *EPiC Comp.*, pages 55–67. EasyChair, March 2016.

[16] S. H. Haeri and S. Schupp. Component-Based Mechanisation of Programming Languages, a Model for. April 2017. Submitted.

[17] P. Herzum and O. Sims. *Business Component Factory: A Comprehensive Overview of Component-Based Development for the Enterprise (OMG)*. Wiley, 1 edition, April 2008. Kindle Edition.

[18] C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic Embedding of DSLs. In Y. Smaragdakis and J. G. Siek, editors, 7^{th} *GPCE*, pages 137–148, Nashville, TN, USA, October 2008. ACM.

[19] A. B. Madsen and E. Ernst. Revisiting Parametric Types and Virtual Classes. In J. Vitek, editor, 48^{th} *TOOLS*, volume 6141 of *LNCS*, pages 233–252. Springer, June 2010.

[20] M. D. McIlroy. Mass Produced Software Components. In *Proc. NATO Conf. Soft. Eng.*, pages 138–155, New York, US, 1969. Petrocelli/Charter.

[21] M. Mernik. An Object-Oriented Approach to Language Compositions for Software Language Engineering. *JSS*, 86(9):2451–2464, 2013.

[22] M. Odersky and M. Zenger. Independently Extensible Solutions to the Expression Problem. In *FOOL*, January 2005.

[23] M. Odersky and M. Zenger. Scalable Component Abstractions. In 20^{th} *OOPSLA*, pages 41–57, San Diego, CA, USA, 2005. ACM.

[24] B. C. d. S. Oliveira. Modular Visitor Components. In 23^{rd} *ECOOP*, volume 5653 of *LNCS*, pages

269–293. Springer, 2009.

[25] B. C. d. S. Oliveira and W. R. Cook. Extensibility for the Masses – Practical Extensibility with Object Algebras. In 26^{th} *ECOOP*, volume 7313 of *LNCS*, pages 2–27. Springer, 2012.

[26] B. C. d. S. Oliveira, T. van der Storm, A. Loh, and W. R. Cook. Feature-Oriented Programming with Object Algebras. In Giuseppe Castagna, editor, 27^{th} *ECOOP*, volume 7920 of *LNCS*, pages 27–51, Montpellier, France, 2013. Springer.

[27] S. L. Pfleeger and J. M. Atlee. *Software Engineering: Theory and Practice*. Pearson, 4 edition, May 2009. International Version.

[28] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In M. Aksit and S. Matsuoka, editors, 11^{th} *ECOOP*, volume 1241 of *LNCS*, pages 419–443, Jyväskylä, Finland, 1997.

[29] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 7 edition, 2009.

[30] X. Qi and A. C. Myers. Sharing Classes between Families. In M. Hind and A. Diwan, editors, *PLDI*, pages 281–292. ACM, June 2009.

[31] T. Rendel, J. I. Brachthäuser, and K. Ostermann. From Object Algebras to Attribute Grammars. In A. P. Black and T. D. Millstein, editors, 28^{th} *OOPSLA*, pages 377–395. ACM, October 2014.

[32] J. C. Reynolds. User-Defined Types and Procedural Data Structures as Complementary Approaches to Type Abstraction. In S. A. Schuman, editor, *NDAL*, pages 157–168. INRIA, 1975.

[33] T. Rompf and M. Odersky. Lightweight Modular Staging: a Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In 9^{th} *GPCE*, pages 127–136, Eindhoven, Holland, 2010.

[34] I. Sommerville. *Software Engineering*. Addison Wesley, 9 edition, 2011.

[35] W. Swierstra. Data Types à la Carte. *JFP*, 18(4):423–436, 2008.

[36] S. Thaker, D. S. Batory, D. Kitchin, and W. R. Cook. Safe Composition of Product Lines. In C. Consel and J. L. Lawall, editors, 6^{th} *GPCE*, pages 95–104, Salzburg, Austria, 2007. ACM.

[37] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comp. Surv.*, 47(1):6:1–6:45, June 2014.

[38] M. Torgersen. The Expression Problem Revisited. In M. Odersky, editor, 18^{th} *ECOOP*, volume 3086 of *LNCS*, pages 123–143, Oslo (Norway), June 2004.

[39] P. Wadler. The Expression Problem. Java Genericity Mailing List, November 1998.

[40] Y. Wang and B. C. d. S. Oliveira. The Expression Problem, Trivially! In 15^{th} *Modularity*, pages 37–41, New York, NY, USA, 2016. ACM.

[41] M. Zenger and M. Odersky. Extensible Algebraic Datatypes with Defaults. In 6^{th} *ICFP*, pages 241–252, Firenze (Florence), Italy, 2001. ACM.