EPiC Series in Computing

Volume 39, 2016, Pages 55–67

SCSS 2016. 7th International Symposium on Symbolic Computation in Software Science

Expression Compatibility Problem

Seyed H. Haeri (Hossein)[1] and Sibylle Schupp[2]

[1] Université catholique de Louvain, Belgium hossein.haeri@uclouvain.be
[2] Hamburg University of Technology, Germany schupp@tuhh.de

Abstract

We solve the Expression Compatibility Problem (ECP) – a variation of the famous Expression Problem (EP) which, in addition to the classical EP concerns, takes into consideration the replacement, refinement, and borrowing of algebraic datatype (ADT) cases. ECP describes ADT cases as components and promotes ideas from Lightweight Family Polymorphism, Class Sharing, and Expression Families Problem (EFP). Our solution is based on a formal model for Component-Based Software Engineering that pertains to EP. We provide the syntax, static semantics, and dynamic semantics of our model. We also show that our model solves EFP as well. Moreover, we show how to embed the model in Scala.

1 Introduction

Expression Problem (EP) [3, 26, 33] is amongst the most famous problems in the community of Programming Languages (PLs). A wide range of EP solutions have thus far been proposed. Consider [18, 21, 30, 32], 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 *strong static type safety*, i.e., applying a function f on a statically 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.

The main reason why EP is peculiarly recurrent in the PLs community is perhaps for it is customary to implement the syntax of a PL as an ADT. Then, the PL semantics as well as the analyses and transformations that pertain to the PL can all comfortably be considered functions on that ADT. As such, EP is prudent about minimising the side effects of extending an ADT with new cases and functions on it. However, in addition to extension, PLs also experience the following *trio*: refinement, replacement, and removal of constructs. For example, C++ extended C with OOP but also refined the C scoping rules. Later on, Java removed multiple-inheritence from C++. Finally, Scala added traits to Java, which borrows from mixins and ML modules. Despite that, the impact of the trio is hardly studied w.r.t. EP.

In this paper, we introduce and solve the *Expression Compatibility Problem* (ECP), which is a variation of EP that, whilst adding extra concerns about extension, takes the trio too into consideration. Components are a cornerstone to ECP. This is because, in order to ensure soundness properties for the trio, one needs to enforce constraints at the level of PL constructs. The implication is modelling PL constructs using components in their Component-Based Software Engineering (CBSE) sense. With its above demand for components that are designed for cross-PL reuse, ECP is also a variation of the Expression Families Problem (EFP) [20]. The next paragraph gives an outline of ECP in terms of the EP terminology. For the rest of this paper, we dismiss the PL implementation concerns and employ the EP terminology.

ECP is the challenge of finding an implementation for an ADT that uses components to address EP and:

EC1. is *independently extensible* [31, 18], i.e., it should be possible to compose independently developed extensions.

EC2. is *backward compatible*, i.e., upon extension, the existing code retains its consistency and remains available for use – specially, to the extension.

EC3. is *scalable* [17], i.e., adding a new case or function takes proportional code.

EC4. ensures *completeness of component composition*, i.e., no valid combination of components should be rejected statically or dynamically.

EC5. *statically* ensures *soundness of component composition*, i.e., provides means to enforce compilation failure for invalid component combinations.

EC6. is *combination sensitive*, i.e., distinguishes between ADT types by their component combinations. In particular, the type identity of a core ADT should be distinguishable from that of its extension.

EC7. *accumulates nominal subtyping*, i.e., recognises the component combination c_1 as a structural subtype of c_2 when every case in c_1 is a nominal subtype of a case in c_2.

EC8. guarantees *syntactic compatibility upon extension*, i.e., statically rejects addition of new cases when the syntactic category of one existing case (or more) is neither retained intact nor refined.

EC1–EC3 and EC8 underpin new extension concerns w.r.t. EP. Refinement and removal concerns are EC4, EC5, EC6, and EC7. Then, replacement is the focus of EC4, EC5, and EC6.

One may wonder why EC8 in presence of EC5? EC8 is particularly concerned about the sanity of extensions; whereas EC5 is also about core combinations ab initio.

Contributions We define ECP and present a solution for it in Section 2. In Section 3, we introduce $\gamma\Phi C_0$ as a formal solution to ECP. In summary, $\gamma\Phi C_0$: facilitates minimal shape exposure using its family parameterisation; accommodates component late-binding using a notion component equivalence; and, features static dependency on the 'requires' interface of its components. By offering solutions to the two EFP exercises [20][1], we show how $\gamma\Phi C_0$ can be used to solve EFP. In Section 5, we demonstrate highlights of how to simulate $\gamma\Phi C_0$ in Scala.

Paper Organsiation We begin by Section 2 where the benefits of an ECP solution are highlighted. Afterwards, in Section 3, a formal presentation of $\gamma\Phi C_0$ is briefed. More particularly, the $\gamma\Phi C_0$ syntax comes in Section 3.1. Selected parts of the $\gamma\Phi C_0$ semantics that are in its front line of solving ECP are discussed in Section 3.2. How solving EFP can be accomplished in $\gamma\Phi C_0$ is, then, shown in Section 4. Section 5 explains how to simulate the $\gamma\Phi C_0$ material of this paper in Scala. Next, we explore the related work in Section 6. Finally, conclusion and future work come in Section 7.

[1] designed for exhibiting the power of Modular Visitor Components (MVCs) for solving EFP

In this paper, we only show how to employ $\gamma\Phi C_0$ for compositional analyses and transformations [11]. For how to employ $\gamma\Phi C_0$ in non-compositional scenarios, see [8, §8]. In addition, we will follow the EP tradition of using basic integer arithmetic as the running example of this paper. To examine the performance of our material under real research languages, the reader is invited to consider [8]. The material that we present in this paper, including both the $\gamma\Phi C_0$ and the Scala code, is available online at: http://www.sts.tu-harburg.de/people/hossein/ecp.html.

2 Once upon a time when ECP was already tackled...

This section aims to explore the potential opportunities that a solution to ECP will create for safe reuse. That is accomplished by a rapid presentation of how $\gamma\Phi C_0$ manages ECP. On our way, we introduce the relevant $\gamma\Phi C_0$ features and the techniques needed to get them to serve our purpose.

Components We begin the show with the introduction of a few ADT cases. $\gamma\Phi C_0$ facilitates that using its special support for *component* definitions:

```
01  component Num<X ◁ Num> {          06      X left, right;
02    int n;                          07      Sub(X left, X right) {
03    Num(int n) {this.n = n;}        08        this.left = left; this.right = right;
04  }                                 09      }
05  component Sub<X ◁ Num ⊕ Sub> {    10    }
```

The code above introduces two components for ADT cases for signed integers (*Num* in lines 01–04), and subtraction (*Sub* lines 05–10). The body of a component is like that of a class in Igarashi et al. [12]: Line 02 above states that *Num* has a field n of type int. Line 03 demonstrates *Num*'s constructor. And, although not shown here, $\gamma\Phi C_0$ components can also define (non-constructor) methods. Hereafter, we assume *UNm* for unsigned integers and *Add* for addition as well. The two latter components can be obtained similarly.

Unlike an Igarashi et al. class, however, a $\gamma\Phi C_0$ component enjoys *family parameterisation* too. This is for a component to express its 'requires' interface ([29, §17] and [23, §10]). For example, line 05 above states that *Sub* 'requires' for its container ADT to contain *Num* and *Sub*. (In this paper, *Sub* represents signed subtraction, exclusively. Hence, it 'requires' the signed integer case – i.e., *Num* – as well.) Inside *Sub*'s body, the *family parameter* X plays the container role and is to be substituted for a *family*. (See below for more.) Built-in types aside, the only types allowed inside a component body are its family parameter (e.g., lines 06 and 07 above) and the components it 'requires'. (For types of the latter group, see lines 02 and 05 in *EN* and *EU* discussed later on.)

```
01  family Φ0 = UNm ⊕ Add;          04  family Φ3 = Num ⊕ Sub;
02  family Φ1 = Num ⊕ Add;          05  family Φ4 = Num ⊕ Add ⊕ Sub;
03  family Φ2 = UNm ⊕ Sub;          06  family Φ5 = Sub;
//Error! Line 03: Sub requires Num too. Line 06: Sub requires Num too.
```

Families, EC4, and EC5 A $\gamma\Phi C_0$ *family* definition can be employed for an ADT introduction. As shown above, a family definition is as simple as listing the components it contains. Families Φ_0, Φ_1, Φ_3, and Φ_4 are examples of $\gamma\Phi C_0$'s support for EC4: The programmer is entitled to mix components in all valid ways. On the contrary, Φ_3 and Φ_5 attempts are statically rejected. Those are examples of $\gamma\Phi C_0$'s support for EC5.

Clients Functions on ADTs are implemented in $\gamma\Phi C_0$ using its so-called *client* definitions. The two differences between $\gamma\Phi C_0$ components and clients are as follows: Firstly, the former can have fields as well as methods, but, the latter is only a pack of methods. Secondly, the former takes a single family parameter, whilst, the latter can take multiple such parameters. Whilst the first difference is displayed in this section, the second will only be discussed in Section 3.

The code below on evaluation of integer arithmetic expressions is an example of how to define functions on ADTs.

```
01  client EN<X ◁ Num> {//signed eval
02    int eval(X.Num x,
↪               int e(X)){return x.n;}
03  }
04  client EU<X ◁ UNm> {//unsigned eval
05    uint eval(X.UNm x,
↪               uint e(X)){return x.n;}
06  }
```

The most noteworthy point about the two clients above is that they only handle their own part of evaluation. Note that, in order to handle a case, explicit nomination of the case is required. The consequence, which was also stated earlier on, is that: any attempt to handle other cases than the ones nominated in the 'requires' interface will be statically rejected. For example, would it happen that the programmer mistakenly chooses to handle subtraction in *EN*, a compile error will be emitted. The second parameter of *eval* (in lines 02 and 05 above) represents the complete evaluation recipe, the role of which will become more clear below.

```
//signed integer and addition eval
01  client ENA<X ◁ Num ⊕ Add>{
02    int eval(X x, int e(X)){
03      return x match {
04        case X.Add ⇒
↪           e(x.left) + e(x.right);
05        case X.Num ⇒
↪           EN<X as Num>.eval(x, e);
06  } } }
```

Consider line 04 above, where the joy of compositional evaluation and component-based development are experienced in tandem: Not having the complete recipe, it leaves evaluation of subexpressions to the parameter *e*. At the same time, only the addition case is handled by that line. The action of line 05 is slightly different, where handling the case (of signed integers) is relayed to the *eval* method of *EN*. More remarkable is the use of family parameter *projection* for the relaying. When not all the components of 'requires' interface are to be passed in the exact same order, $\gamma\Phi C_0$ demands explicit nomination of the relevant ones. Otherwise, the family parameter alone can be nominated. (See $Eval_1$ and $Eval_4$ below.)

EC7 and EC8 Line 05 above also demonstrates another important property of $\gamma\Phi C_0$, i.e., how it addresses EC7: It is legal for *ENA* to reuse the *EN* methods for its 'requires' interface is a superset of that of *EN*. The dual of that is how the following piece of code fails statically due to $\gamma\Phi C_0$'s implementation of EC8.

```
01  client BadENA<X ◁ Num ⊕ Add> {
02    ... EU<???>.eval(...);//Error! Invalid projection.
03  }
```

The reader is invited to take their time to observe that no valid combination of the components in the 'requires' interface of *BadENA* can be substituted for ??? in line 02.

EC1 Due to its similarity, we drop the definition of *ENS*<*X* ◁ *Num* ⊕ *Sub*> to move to the demonstration of how $\gamma\Phi C_0$ addresses EC1 as a function concern. The lines 04 and 05 below show how *ENAS* composes independently developed extensions of *EN*, i.e., *ENA* and *ENS*. (It is trivial to observe that $\gamma\Phi C_0$'s family definition facility addresses EC1 as a case concern.)

```
01  client ENAS<X ◁ Num ⊕ Add ⊕ Sub> {
02    int eval(X x, int e(X)){
03      return x match {
04        case X.Add ⇒
↪   ENA<X as Num ⊕ Add>.eval(x, e);
05        case X.Sub ⇒
↪   ENS<X as Num ⊕ Sub>.eval(x, e);
06        case X.Num ⇒
↪   EN<X as Num>.eval(x, e);
07  } } }
```

Tests In order to bring the above clients to action for the defined families, one first ties the recursive knot. $Eval_1$ and $Eval_4$ below demonstrate that for Φ_1 and Φ_4, respectively:

```
01  client Eval1<X ◁ Num ⊕ Add>{
02    int do_it(X x){
03      return ENA<X>.eval(x, do_it);
04  } }
05  client Eval4<X ◁ Num ⊕ Add ⊕ Sub>{
06    int do_it(X x){
07      return ENAS<X>.eval(x, do_it);
08  } }
```

Then, one instantiates the clients using the families defined earlier. Here are some tests:

```
01  val tpf_1 = new Add<Φ1>(new Num<Φ1>(3), new Num<Φ1>(5));
02  val tpf_3 = new Add<Φ3>(new Num<Φ3>(3), new Num<Φ3>(5));//Error! Φ3 doesn't provide Add.
03  Eval_1<Φ1>.do_it(tpf_1); //Returns 8.
04  val tpf_4 = ... //3 + 5 for Φ4
05  val tpfmo_4 = new Sub<Φ4>(tpf_4, new Num<Φ4>(1));
06  Eval_4<Φ4>.do_it(tpf_4); //Returns 8.
07  Eval_1<Φ4>.do_it(tpf_4); //Error! Sub case of Φ4 unmatched by Eval_1.
08  Eval_4<Φ4>.do_it(tpfmo_4); //Returns 7.
09  Eval_1<Φ4>.do_it(tpfmo_4); //Error! Sub case of Φ4 unmatched by Eval_1.
10  Eval_1<Φ4 as Add ⊕ Num>.do_it(tpf_4); //Returns 8.
11  Eval_1<Φ4 as Add ⊕ Num>.do_it(tpfmo_4);//Error! tpfmo_4 has other Φ4 cases than Num and Add.
```

Other ECP Concerns We end by briefly explaining $\gamma\Phi C_0$ on the remaining ECP concerns. The support for EC2 is obvious for addition of new components and clients has no effect on existing programs. $\gamma\Phi C_0$ does also clearly support EC3: Addition of a new case amounts to defining the respective single component; addition of a new function on n cases (e.g., pretty-printing) amounts to defining (at most) n new clients for the corresponding cases and a single client to tie the recursive knot. Understanding that $\gamma\Phi C_0$ addresses EC6 is also easy: Defining a new ADT (e.g., an extension to an existing one) is a matter of defining a new $\gamma\Phi C_0$ family. Such an addition influences no existing family (especially, that of the core ADT, if any). A family is distinguishable from other families (including its core, if any) by its name. On the contrary, clients are equally (un-)available to families with identical component combinations.

Remark 1. *Although after tying the knot for $Eval_1$ or $Eval_4$, they are no longer extensible, that is not a failure in addressing EP. The reason is that the service that was available to Φ_1 and Φ_4 essentially come from ENA and ENAS, which remain extensible upon tying the knot.*

3 Excerpts from $\gamma\Phi C_0$: A Formal Solution to ECP

As a PL, $\gamma\Phi C_0$ is highly inspired by the popular informal models of CBSE. $\gamma\Phi C_0$ is, however, especially tuned for ECP. The opening discussion of this section will delve into the design choices made accordingly. From a PL designer's perspective, on the other hand, one would be interested in finding out about the axes along which to tweak the PL for reuse. In the favour of that interest, over the opening discussion, we further elaborate on the two sources of polymorphism in $\gamma\Phi C_0$. After the opening discussion, selected parts of the $\gamma\Phi C_0$ syntax and semantics are presented in Sections 3.1 and 3.2. For a complete account of $\gamma\Phi C_0$, consult [10]. We now move to the opening discussion itself.

The $\gamma\Phi C_0$ world has three major role-players: components, families, and clients. A $\gamma\Phi C_0$ component takes after its well-known CBSE identity by specifying its 'requires' and 'provides' interfaces. It specifies its 'requires' interface using family parameterisation. The 'provides' interface of a $\gamma\Phi C_0$ component is specified just like a familiar OOP class. $\gamma\Phi C_0$ families are simply defined as a collection of their respective components. Clients in $\gamma\Phi C_0$ are pieces of code that are applicable to any family, provided that the family contains the specified minimal component combination. This is the first source of polymorphism in $\gamma\Phi C_0$: A client (or component) code can be reused amongst all such families (but, not other ones). We refer to this property as the *minimal shape exposure*. Again, $\gamma\Phi C_0$ uses family parameterisation to that end. Minimal shape exposure is particularly geared towards EC4, EC5, and EC6.

Polymorphism in $\gamma\Phi C_0$ comes from another source as well: *component late-binding*. The family parameterisation used for a $\gamma\Phi C_0$ component or client enforces the availability of certain

components. Instead of providing the exact requested components, however, the family is free to mix an *equivalent* component in. As such, the exact behaviour of a family-parameterised code is not known until the actual family used for instantiation is provided. This is our notion of component late-binding. This source of polymorphism serves EC7 and EC8 most specifically.

By specifying its 'requires' interface using family parameterisation, a $\gamma\Phi C_0$ client (or component) determines the component combination it expects from the family that is going to use it. The client (or component) code, then, will be statically bound to the requested combination. Inside the body of a client (or component), any attempt to access unrequested components is outlawed statically. We refer to this feature as static dependency on the 'requires' interface.

3.1 The $\gamma\Phi C_0$ Syntax

D_γ	::= component γ<$X \lhd \oplus\overline{\gamma} \not\lhd \overline{\gamma}$> $\lhd \{\overline{R}\,\overline{f}\,;\ K\ \overline{m_\gamma}\}$	component definition
D_C	::= client C<$\overline{X}\ \lhd \overline{\oplus\overline{\gamma}}$>$\{\overline{m_C}\}$	client definition
D_Φ	::= family $\Phi = \oplus\overline{\gamma}$	family definition
τ	::= $\overline{D_\Phi}\,; I.m_C(\overline{v})$	test

Above comes part of the $\gamma\Phi C_0$ syntax. γ ranges over components, C over clients, K over constructors, m over methods, e over expressions, f over fields, Φ over families, and X over family parameters. Priming a syntactic metavariable or subscripting it with numbers does not change its syntactic category. When distinction between a metavariable of a component and that of a client is required, we use subscripts γ and C. When referring to both categories collectively, we drop the subscript. For example, m_γ and m_C denote component methods and client methods, respectively, but m can be either an m_γ or an m_C. Note that neither subscript is related to a particular component γ or client C instance. Following the tradition of lightweight family polymorphism (Section 6), the overline notation is used for a list of entities and #(.) for the length of a list. For example, for some known n: $\overline{\gamma}$ denotes $\gamma_1\gamma_2\ldots\gamma_n$. The notation $\overline{\gamma}$ is also overloaded to mean the list $\gamma_1, \gamma_2, \ldots, \gamma_n$, when appropriate. We extend that notation for $\oplus\overline{\gamma}$ to mean $\gamma_1 \oplus \gamma_2 \oplus \ldots \gamma_n$. We overload the notation one further step to mean $X_1 \lhd \oplus\overline{\gamma}_1, X_2 \lhd \oplus\overline{\gamma}_2, \ldots, X_n \lhd \oplus\overline{\gamma}_n$ by $\overline{X} \lhd \overline{\oplus\overline{\gamma}}$. Both clients and components take family parameters. The notation $X \lhd \oplus\overline{\gamma}$ stipulates that the family to be substituted for X has to include components (that are equivalent to) $\gamma_1, \gamma_2, \ldots, \gamma_n$. In such a case, we say that $\oplus\overline{\gamma}$ is the *upper bound* of X. When γ is an upper bound of X, the relative type $X.\gamma$ – read the component γ of X – is the component substituted for γ when substituting a family for X.

The syntax for introduction of a family is as simple as enumerating the components it combines, interleaved by "$\oplus$". To test a client on a family, one calls a method of a client by passing appropriate arguments to the method. To that end, either the whole family is used for client instantiation ($C\texttt{<}\cdots, \Phi, \cdots\texttt{>}$) or a *projection* of it ($C\texttt{<}\cdots, \Phi$ as $\oplus\,\overline{\gamma}, \cdots\texttt{>}$). Note that, unlike components, we choose clients to store no data and simply act as a collection of methods. As such, clients need no constructors. A test τ contains a sequence of family introductions and a single call to a method of a client instantiated by the introduced families (or their projections). A $\gamma\Phi C_0$ program is a test along with the components and clients that it uses.

3.2 How does $\gamma\Phi C_0$ address ECP?

This section gives a very short overview of the parts of the $\gamma\Phi C_0$ static semantics that help it solve ECP: the rules (T-Invk$_3$) and (WF-Family) below. Due to space constraints, we will only explain those two rules in the context of how they address a selection of the ECP concerns: EC4, EC5, EC7, and EC8. The interested reader is referred to [10] for more.

$$\frac{\begin{array}{c}\Gamma; C \vdash \overline{e} : \overline{R_e} \qquad \mathit{fps}(C') = \overline{X'} \qquad \#\overline{X'} = \#\overline{S} \\ C \vdash \mathit{ssfp}(\overline{S}, \overline{X'}, C') \text{ ok} \qquad \mathit{mtype}(m_C, C') = \overline{R'} \to R' \\ \overline{R} = \overline{R'}[C \xrightarrow{[\overline{S}/\overline{X'}]} C'] \qquad C \vdash \overline{R_e} <: \overline{R} \qquad R = R'[C \xrightarrow{[\overline{S}/\overline{X'}]} C']\end{array}}{\Gamma; C \vdash C'\texttt{<}\overline{S}\texttt{>}.m_C(\overline{e}) : R} \text{ (T-Invk}_3\text{)}$$

$$\frac{\overline{\gamma} = \mathit{fpub}^*(\Phi) \qquad \mathit{sat\text{-}by}(\overline{\gamma}, \Phi) \qquad \mathit{non\text{-}conf}(\overline{\gamma})}{\mathsf{family}\ \Phi = \cdots \text{ ok}} \text{ (WF-Family)}$$

The clauses used in the above rules are likely to look extraordinarily compact to the untrained eye. Before we get into the intuition behind each rule, we would like to invite the reader to check the names of the helper functions used: *fps* = *family parameters*, *ssfp* = *substitution* of family parameter *selection* for *family parameter*, *mtype* = *method type*, *fpub* = *family parameter upper bound*, *sat-by* = *satisfied by*, and *non-conf* = *non-conflicting*. We would also like to invite them to follow the comprehensions used:

$\Gamma; C \vdash \overline{e} : \overline{R_e}$	for	$\Gamma; C \vdash e_1 : R_{e_1}, \ldots, \Gamma; C \vdash e_n : R_{e_n}$,
$C \vdash \mathit{ssfp}(\overline{S}, \overline{X'}, C')$ ok	for	$C \vdash \mathit{ssfp}(S_1, X'_1, C')$ ok, $\ldots, C \vdash \mathit{ssfp}(S_n, X'_n, C')$ ok,
$C \vdash \overline{R_e} <: \overline{R}$	for	$C \vdash R_{e_1} <: R_1, \ldots, C \vdash R_{e_n} <: R_n$,
$R'[C \xrightarrow{[\overline{S}/\overline{X'}]} C']$	for	$((R'[C \xrightarrow{[S_1/X'_1]} C']) \cdots [C \xrightarrow{[S_n/X'_n]} C'])$, and
$\overline{R} = \overline{R'}[C \xrightarrow{[\overline{S}/\overline{X'}]} C']$	for	$R_1 = R'_1[C \xrightarrow{[\overline{S}/\overline{X'}]} C'], \ldots, R_n = R'_n[C \xrightarrow{[\overline{S}/\overline{X'}]} C']$.

With its number of premises the rule (T-Invk$_3$) might look daunting at the first sight. Below, we explain it informally in a top-down and left-to-right fashion. The rule states that, within the body of a client C, in order for a call to a method m_C of C' to return a value of type R: The argument types $\overline{R_e}$ need to be determined ($\Gamma; C \vdash \overline{e} : \overline{R_e}$); the passed selections $\overline{S}$ to C' need to have the right number ($\mathit{fps}(C') = \overline{X'}$ and $\#\overline{X'} = \#\overline{S}$); the pass of all selections $\overline{S}$ from C to their same-indexed family parameter amongst $\overline{X'}$ of C' needs to be valid ($C \vdash \mathit{ssfp}(\overline{S}, \overline{X'}, C')$ ok); the parameter types $\overline{R'}$ of m_C in C' need to be adapted to C for $\overline{R}$ to be obtained ($\overline{R} = \overline{R'}[C \xrightarrow{[\overline{S}/\overline{X'}]} C']$); the argument types $\overline{R_e}$ need to all be subtypes of (or equal to) their same-indexed $\overline{R}$ type ($C \vdash \overline{R_e} <: \overline{R}$); and, the return type R' of m_C in C' needs to be adapted for R to be obtained ($R = R'[C \xrightarrow{[\overline{S}/\overline{X'}]} C']$). The rule (WF-Family) states that for a family Φ to be well-formed: The components $\overline{\gamma}$ that are directly or indirectly requested by the components combined to obtain Φ – i.e., $\mathit{fpub}^*(\Phi)$ – must all (either themselves or an equivalent of theirs) exist in Φ ($\mathit{sat\text{-}by}(\overline{\gamma}, \Phi)$); and, that there must be no conflict between the $\overline{\gamma}$ (namely, $\mathit{non\text{-}conf}(\overline{\gamma})$). Here is a sketch of how the above rules address the promised concerns.

EC4. The respective rule of this concern is (WF-Family), with the key role player being $\mathit{sat\text{-}by}(\overline{\gamma}, \Phi)$. This clause pronounces Φ sound so long as it does provide all the requested components (or equivalents of theirs).

EC5. Again, the respective rule here is (WF-Family). Albeit, the role of $\mathit{non\text{-}conf}(\overline{\gamma})$ is more important for this concern. This clause makes sure that there is no conflict between the components combined to produce a family.

EC7. The respective rule of this concern is (T-Invk$_3$). It is the $C \vdash \mathit{ssfp}(\overline{S}, \overline{X'}, C')$ ok clause in the premises of that rule that is the most important here. The explicit correspondence between $\overline{S}$ and $\overline{X'}$ is the evidence submitted (by the programmer) for C to be a structural subtype of C'. As such, the reuse of the method m_C of C' by C is authorised by that clause.

For every family parameter X'_i of C', the above clause checks whether the substitution of the selection S_i of C is valid for X'_i of C'.

EC8. The respective rule and key clause of this concern are the same as those of the previous concern. In order to explain how this concern is addressed, however, we need to zoom into $C \vdash ssfp(\overline{S}, \overline{X'}, C')$ ok. Generally, a clause $C \vdash ssfp(X \text{ as } \oplus\overline{\gamma}, X, C')$ ok uses a premise $valid\text{-}as(C, X, \oplus\overline{\gamma})$ that ensures the following: Projection of a family parameter X of C to $\oplus\overline{\gamma}$ is valid. As such, it will reject the reuse of the method m_C of C' otherwise.

Note that we write $sat\text{-}by(\gamma, \Phi)$ when $\exists\gamma' \in \Phi.\ \gamma \overset{s}{\equiv} \gamma'$. (The symbol "$\overset{s}{\equiv}$" used here is our notion of component *equivalence*, which we will not delve into in this paper.) In words, $sat\text{-}by(\gamma, \Phi)$ checks whether Φ does indeed mix a component (that is equivalent to) γ.

4 Expression Families Problem

In his seminal work on EFP [20], Oliveira provides two self-contained examples that obviate the necessity of solving EFP and the strength of MVCs: Equality Tests and Narrowing. We claimed in the Introduction that ECP is a variation of EFP. The first goal of this section is to substantiate that claim by offering $\gamma\Phi C_0$ solutions to Equality Tests and Narrowing in Sections 4.1 and 4.2, respectively. The second goal of this section is to get the reader to observe how straightforward of a solution to EFP a solution to ECP is.

4.1 Equality

The purpose of the Equality Test exercise is to provide a statically safe solution for *multiple dispatching* that is also extensible. Like that of Oliveira, we offer a solution that simulates multi-methods [1, 2]. The driving example is structural equality between expressions. The test on *Num*, *Add*, and *Sub* is performed as follows: (1) $equal(Num(n), Num(n')) = (n == n')$, (2) $equal(Add(e_1, e_2), Add(e'_1, e'_2)) = equal(e_1, e'_1) \wedge equal(e_2, e'_2)$, (3) $equal(Sub(e_1, e_2), Sub(e'_1, e'_2)) = equal(e_1, e'_1) \wedge equal(e_2, e'_2)$, and (4) $equal(\_, \_) = \bot$.

Such a formula is typically implemented as a pattern matching on the two expressions together. Using a familiar pattern matching, the implementation will, however, lose extensibility. To prevent that loss, our solution is first to **decentralise** the pattern matching by implementing each case using a single client. Then, one **integrate**s as many of such clients as appropriate in another client, which also ties the recursive knot. *EqNum* below is the equality client that corresponds to (1):

```
01  client EqNum<X ◁ Num> {
02    bool eq(X.Num x1, X.Num x2, bool e(X, X)){return x1.n == x2.n;}
03  }
```

Note that both *EqNum* only takes care of its own part of decentralisation. In a similar manner, one implements *EqAdd*$<X \lhd Add>$, *EqSub*$<X \lhd Sub \oplus Num>$, and *EqDef*$<X \lhd \epsilon>$ for the other cases. Due to minimal shape exposure of X, none of the above four equality clients sees more than its pertinent part of the recipe. Nevertheless, for usage at the appropriate point (e.g., line 2 above), provisional access to the complete recipe is granted to them via their parameter e. The complete recipe itself can only be obtained upon tying the recursive knot, which referred to as the integration. Eq_1 below integrates the appropriate clients for Φ_1:

```
01  client Eq1<X ◁ Num ⊕ Add>{
02    bool equal(X x1, X x2) {
03      return (x1, x2) match {
04        case (X.Num, X.Num) ⇒ EqNum<X as Num>.eq(x1, x2, equal);
```

```
05          case (X.Add, X.Add) ⇒ EqAdd<X as Add>.eq(x1, x2, equal);
06          case _ ⇒ EqDef<X as ϵ>.eq(x1, x2, equal);
07    } } }
```

Likewise, one implements Eq_4 to integrate clients for Φ_4 and perform the following tests:

```
01  val tpfive1 = ... //3 + 5 for Φ1    03  Eq1<Φ1>.equal(tpfive1, tpfour1) //Returns ⊥.
02  val tpfour1 = ... //3 + 4 for Φ1    04  Eq4<Φ4>.equal(tpf4, tpfmo4) //Returns ⊥.
```

Observe how the technique caters for extensibility by leaving open the possibility of mixing-in new equality clients without the need to touch the existing equality cases. Note that, in this very case, the structural exercise happened to come with a default case. Section 4.2 contains an example where our solution works in the absence of default cases.

4.2 Conversion and Narrowing

Following Oliveira [20], 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 PL design community to provide extensions to a core PL such that the extension programs would then be narrowed to the core (for evaluation and the like). Oliveira shows how his MVCs can be leveraged in favour of correctness for narrowing as a static guarantee that the result of this process will not contain instances from 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. In particular, we will craft a conversion $[\![.]\!]$ from an ADT with subtraction to an ADT with addition and negation (of signed integers): $[\![e_1 - e_2]\!] = [\![e_1]\!] + (-([\![e_2]\!]))$. We assume a component $Neg\langle X \lhd Num \oplus Neg\rangle$ for negation. Then, we implement the decentralised pattern matching clients: $N2N\langle X_1 \lhd Num, X_2 \lhd Num\rangle$ to convert Num to Num, $G2G\langle X_1 \lhd Num \oplus Neg,\ X_2 \lhd Num \oplus Neg\rangle$ to convert Neg to Neg, and $A2A\langle X_1 \lhd Add,\ X_2 \lhd Add\rangle$ to convert Add to Add. The only non-identical conversion is that of subtraction expressions to the equivalent combination of addition and negation:

```
01  client S2AN<X1 ◁ Num ⊕ Sub, X2 ◁ Num ⊕ Add ⊕ Neg> {
02    X2 convert(X1.Sub x1, X2 c(X1)) {
03      return new Add<X2>(c(x1.left), new Neg<X2>(c(x1.right)));
04  } }
```

In the snippet above, X_1 and X_2 are the family parameters corresponding to the first and the second ADT, respectively. Furthermore, the parameter c of *convert* in line 02 is the function that devises the complete conversion recipe. The integration is not largely new:

```
01  client ConvSub<X1 ◁ Num ⊕ Add ⊕ Sub ⊕ Neg, X2 ◁ Num ⊕ Add ⊕ Neg> {
02    X2 doit(X1 x1) {
03      return x1 match {
04        case X1.Num ⇒ N2N<X1 as Num, X2 as Num>.convert(x1, doit);
05        case X1.Neg ⇒ G2G<X1 as Num ⊕ Neg,
↪                          X2 as Num ⊕ Neg>.convert(x1, doit);
06        case X1.Add ⇒ A2A<X1 as Add, X2 as Add>.convert(x1, doit);
07        case X1.Sub ⇒ S2AN<X1 as Num ⊕ Sub,
↪                            X2 as Num ⊕ Add ⊕ Neg>.convert(x1, doit);
08  } } }
```

Narrowing would, then, be a simple application of the above conversion from an ADT to the appropriate projection of the same ADT:

```
01  client NarrowSub<X ◁ Num ⊕ Add ⊕ Sub ⊕ Neg> {
02    X doit(X x) {
03      return ConvSub<X, X as Num ⊕ Add ⊕ Neg>.doit(x);
04  } }
```

Supposing the availability of the two families below

01 family $\Phi_6 = Num \oplus Add \oplus Sub \oplus Neg$; 02 family $\Phi_7 = Num \oplus Add \oplus Neg$;
finally, can one perform the following tests:
val $tpfmone_6 = \ldots$ //$(3+5)-1$ for Φ_6
$ConvSub$<Φ_6, Φ_7>.$doit(tpfmone_6)$ //Returns $(3+5)+(-(1))$ for Φ_7.
$NarrowSub$<Φ_6>.$doit(tpfmone_6)$ //Returns $(3+5)+(-(1))$ for Φ_6.

5 Implementation

Whilst $\gamma\Phi C_0$ still has no dedicated implementation (say a stand-alone compiler and an IDE), an embedding of its is indeed possible in Scala. Details of how to set up a complete Scala codebase for this purpose can be found in our earlier works: [9] and [8, §7 and §8]. In this section, we only explain what Scala code corresponds to what $\gamma\Phi C_0$ element of Section 2.

The presentations in this paper are all in Scala. However, the same solutions can be used in any host PL providing multiple inheritance and type constraints. We use multiple inheritance for our cases to state their syntactic categories in addition to the ADT they are a case of. Type constraints are used as our means for checking soundness of component combinations. The Scala elements that we use here are those initially presented in [19]. Our implementation, however, is rather inspired by Lightweight Modular Staging (LMS) [27]. From another viewpoint, the implementation can also be regarded as a Scala simulation for Polymorphic Variants [6].

We start by the correspondent of a $\gamma\Phi C_0$ component: an ordinary Scala class with specially-wired type parameterisation. Below is the `Sub` class for the Section 2 *Sub* component.

```
1 class Sub[
2   E <: IAE[E], N <: Num[E, N] with E, S <: Sub[E, N, S] with E
3 ](left: E, right: E)
```

Recall that the family parameterisation of *Sub* was $X \lhd Num \oplus Sub$. The translation of that is what comes in line 2 above. In their order of appearance: `E` represents X, `N` represents $X.Num$, and `S` represents $X.Sub$. In Scala, the uses of `<:` in line 2 above demand nominal subtyping. (More on `IAE` later.) Part of the verbosity in the type annotations in line 2 above is because of the idiosyncrasies of JVM that dictate similar F-Boundings to the Scala type system in return of correctness.

```
1 trait Phi4 extends IAE[Phi4]
2 case class Num4(n: Int) extends Num[Phi4, Num4](n) with Phi4
3 case class Add4(left: Phi4, right: Phi4) extends ...
4 case class Sub4(left: Phi4, right: Phi4) extends ...
```

The Scala counterpart of the Φ_4 definition is even more verbose. The trait `Phi4` in line 1 above is like the "family Φ_4" part of the family definition. `IAE` is our interface for integer arithmetic expressions. An ADT that is to use the arithmetic expression cases needs to tie the F-Bound knot of `IAE` in a similar fashion. Lines 2 to 4 add the *Num*, *Add*, and *Sub* cases to `Phi4`, respectively. Notice how, unlike $\gamma\Phi C_0$ in which Φ_4 simply bundles its appropriate components together, in Scala, one cannot reuse the exact same component names for the cases of `Phi4`. For instance, the code above calls the *Add* case `Add`**4** (as opposed to just `Add`).

```
1 class ENAS[
2   E <: IAE[E],            N <: Num[E, N] with E,
3   A <: Add[E, A] with E, S <: Sub[E, N, S] with E
4 ] {
5   def eval(x: E, e: E => Int): Int = x match {
6     case _: Add[_, _] => new ENA[E, N, A].eval(x, e)
```

```
7       ...
8   }   }
```

The code above is the Scala counterpart for *ENAS* of Section 2. For the first difference, note the "`_:`" before `Add` at line 6. In Scala, in addition to their types, patterns need names.

```
1   object test2 {
2     val three_plus_five4 = Add4(Num4(3), Num4(5))
3     def run1 = new Eval4[Phi4, Num4, Add4, Sub4].do_it(three_plus_five4)
4   }
```

Finally, `three_plus_five4` above constructs a corresponding expression for $3+5$ in Φ_4. Note the interesting advantage gained by the tight binding of `Add4` to `Phi4`: Unlike *tpf*$_4$ that requires explicit mention of Φ_4 at every component instantiation, explicit mention of `Phi4` is absent in `Add4` and `Num4` uses in line 2. On the contrary, for `Phi4` has no knowledge about its cases, in line 3 above, one needs to redundantly list the cases of `Phi4` upon instantiation of `Eval4` for it. Contrast that to $\gamma\Phi C_0$ version where the sole mention of Φ_4 suffices.

6 Related Work

Family Polymorphism [4, 5] aims to ensure that certain interrelationships between the constituents of a system – referred altogether as a *family* – are all extension-invariant. Support for EC1 and EC2 is clear. Given its difficulties in cross-family class sharing [15], whether Family Polymorphism can also support EC3 is hard to judge. With its absence of a clear counterpart for components, we cannot assess how addressable the remaining ECP concerns are.

Jx [16] starts a series of works on sharing nested classes amongst families. J& [17] generalises to intersection types and extension composition (cf. EC1). Of the ECP concerns, it also addresses EC3 and EC2. In $J\&_s$ [24], sharing of classes amongst families is nomination-based but not exclusively hierarchical. $J\&_h$ [25] moves to **homogeneous** class sharing, where a core family and all its extensions are *equivalent*, probably risking EC2.

Having observed the heavy workload of family polymorphism in certain circumstances, .FJ [28] presents lightweight family polymorphism, which addresses EC2 and EC3. It does not address EC1 because .FJ only supports extension using single (nominal) inheritance of top-level families. For its lack of counterpart for components, the rest of ECP concerns are irrelevant to .FJ. Then, FGJ# [13] presents *type parameter members* to alleviate the monolithicity of (lightweight) family polymorphism in that family members are inseparable from their enclosing families. Whether FGJ# supports EC4, EC7, and EC8 is not known. On the other hand, for similar reasons to .FJ, FGJ# addresses EC2 and EC3 but not EC1. Its move towards components shapes an interesting behaviour w.r.t. the rest of ECP concerns. It can simulate them all so long as an ADT interface hard-wires the relevant requirements. X.FGJ [14] is the next work of this group. X.FGJ is in remarkable proximity to $\gamma\Phi C_0$with its member interfaces and families, which take one further step toward getting component-based. Nevertheless, X.FGJ scores alike FGJ# regarding the ECP concerns.

Oliveira was the first to frame EFP. He also offered MVCs [20] as an EFP solution. Oliveira and Cook [21] back this work up by the powerful and simple concept of *object algebras*. Later, Oliveira et al. [22] try to address awkwardness issues faced in their former paper upon composition of object algebras. Which element in the latter two works to compare against $\gamma\Phi C_0$ components (and families) is not straightforward to us. With that uncertainty, here, we only concentrate on Oliveira's seminal work. MVCs address all EC concerns except EC3, EC5, and EC8, although, their support for EC7 is doubtful for its reverse direction [8].

Of the rich literature on EP we only consider a couple that we deem close enough to this paper. Polymorphic Variants [6, 7] solve EP using global case definitions that, at their point of definition, become available to every ADT defined afterwards. They address all the ECP concerns except EC5 and EC6. Yet, from ECP's standpoint, their most important unavailable factor is a notion of components. Rompf and Odersky [27] employ a fruitful combination of the Scala features to present a very simple yet effective solution to EP using LMS. The support of LMS for E2 can be easily broken using an incomplete pattern matching, and is debatable. Although LMS is not based on components, it scores all the ECs except EC5, EC6, and EC8.

7 Conclusion and Future Work

In this paper, we introduce ECP by formulating its eight concerns (Section 1). We show the benefits of solving ECP along with a technology used to solve it (Section 2). We present the syntax and semantics of $\gamma\Phi C_0$ as a formal solution to ECP. We summarise what key role the $\gamma\Phi C_0$ semantics plays in solving ECP (Section 3.2). Furthermore, we show how $\gamma\Phi C_0$ can be used for a solution to EFP (Section 4). Next, we explain how to simulate $\gamma\Phi C_0$ in Scala (Section 5). Finally, the relevant literature is discussed in Section 6.

The immediate future work to this paper is proving the standard theoretical results (e.g., subject reduction, progress, and type soundness) about $\gamma\Phi C_0$. As stated in the introduction, ECP is a variation to EP. As such, $\gamma\Phi C_0$ is the first PL with a formal semantics that is designed for solving EP. It is tempting to try formal proofs for that claim. An extended future work would, then, be flourishing Section 3.2 to formal proofs for $\gamma\Phi C_0$ solving ECP. A fresh look into the paper reveals that it also introduces a new variation on family polymorphism that we would call *component family polymorphism*. Implementing $\gamma\Phi C_0$ is the other obvious future work of this paper. That would form an interesting test bed for component family polymorphism.

References

[1] C. Chambers and G. T. Leavens, *Typechecking and Modules for Multimethods*, TOPLAS **17** (1995), no. 6, 805–843.

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

[3] W. R. Cook, *Object-Oriented Programming Versus Abstract Data Types*, Proc. FOOL (Noordwijkerhout (The Netherlands)) (J. W. de Bakker, W. P. de Roever, and G. Rozenberg, eds.), LNCS, vol. 489, 1990, pp. 151–178.

[4] E. Ernst, *Family Polymorphism*, Proc. 15^{th} ECOOP (J. Lindskov Knudsen, ed.), LNCS, vol. 2072, Springer, June 2001, pp. 303–326.

[5] ———, *Reconciling Virtual Classes with Genericity*, Proc. 7^{th} JMLC (D. E. Lightfoot and C. A. Szyperski, eds.), LNCS, vol. 4228, Springer, September 2006, pp. 57–72.

[6] J. Garrigue, *Programming with Polymorphic Variants*, Proc. 5^{th} W. ML (X. Leroy and D. MacQueen, eds.), Baltimore, MD, USA, September 1998.

[7] ———, *Code Reuse through Polymorphic Variants*, W. Found. Soft. Eng., no. 25, 2000, pp. 93–100.

[8] S. H. Haeri, *Component-Based Mechanisation of Programming Languages in Embedded Settings*, Ph.D. thesis, Inst. Soft. Sys., Hamburg U. Tech., Germany, December 2014.

[9] S. H. Haeri and S. Schupp, *Reusable Components for Lightweight Mechanisation of Programming Languages*, Proc. 12^{th} SC (W. Binder, E. Bodden, and W. Löwe, eds.), LNCS, vol. 8088, Springer, June 2013, pp. 1–16.

[10] ———, *Expression Compatibility Problem – Extended Version*, Technical Report, Inst. Soft. Sys., Hamburg U. Tech., June 2014, available online at http://archive.org/edit/ExpressionCompatibilityProblem.

[11] C. Hofer and K. Ostermann, *Modular Domain-Specific Language Components in Scala*, Proc. 9^{th} GPCE (Eindhoven, The Netherlands) (E. Visser and J. Järvi, eds.), ACM, 2010, pp. 83–92.

[12] A. Igarashi, B. C. Pierce, and P. Wadler, *Featherweight Java: A Minimal Core Calculus for Java and GJ*, TOPLAS **23** (2001), no. 3, 396–450.

[13] T. Kamina and T. Tamai, *A Design and Implementation of Lightweight Constructs for Mutually Extensible Components*, (2008), Submitted Jan. 2008 to Elsevier.

[14] ———, *Lightweight Dependent Classes*, Proc. 7^{th} GPCE (Nashville, TN, USA) (Y. Smaragdakis and J. G. Siek, eds.), ACM, October 2008, pp. 113–124.

[15] A. B. Madsen and E. Ernst, *Revisiting Parametric Types and Virtual Classes*, Proc. 48^{th} TOOLS (J. Vitek, ed.), LNCS, vol. 6141, Springer, June 2010, pp. 233–252.

[16] N. Nystrom, S. Chong, and A. C. Myers, *Scalable Extensibility via Nested Inheritance*, Proc. 19^{th} OOPSLA (J. M. Vlissides and D. C. Schmidt, eds.), ACM, October 2004, pp. 99–115.

[17] N. Nystrom, X. Qi, and A. C. Myers, *J&: Nested Intersection for Scalable Software Composition*, Proc. 21^{st} OOPSLA (Portland, Oregon, USA), ACM, 2006, pp. 21–36.

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

[19] ———, *Scalable Component Abstractions*, Proc. 20^{th} OOPSLA (San Diego, CA, USA), ACM, 2005, pp. 41–57.

[20] B. C. d. S. Oliveira, *Modular Visitor Components*, Proc. 23^{rd} ECOOP, LNCS, vol. 5653, Springer, 2009, pp. 269–293.

[21] B. C. d. S. Oliveira and W. R. Cook, *Extensibility for the Masses – Practical Extensibility with Object Algebras*, Proc. 26^{th} ECOOP, LNCS, vol. 7313, Springer, 2012, pp. 2–27.

[22] B. C. d. S. Oliveira, T. van der Storm, A. Loh, and W. R. Cook, *Feature-Oriented Programming with Object Algebras*, Proc. 27^{th} ECOOP (Montpellier, France) (Giuseppe Castagna, ed.), LNCS, vol. 7920, Springer, 2013, pp. 27–51.

[23] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, 7^{th} ed., McGraw-Hill, 2009.

[24] X. Qi and A. C. Myers, *Sharing Classes between Families*, Proc. PLDI (M. Hind and A. Diwan, eds.), ACM, June 2009, pp. 281–292.

[25] ———, *Homogeneous Family Sharing*, Proc. 25^{th} OOPSLA (Reno/Tahoe, Nevada, USA) (W. R. Cook, S. Clarke, and M. C. Rinard, eds.), ACM, October 2010, pp. 520–538.

[26] J. C. Reynolds, *User-Defined Types and Procedural Data Structures as Complementary Approaches to Type Abstraction*, New Dir. in Algo. Lang. (S. A. Schuman, ed.), 1975, pp. 157–168.

[27] T. Rompf and M. Odersky, *Lightweight Modular Staging: a Pragmatic Approach to Runtime Code Generation and Compiled DSLs*, Proc. 9^{th} GPCE (Eindhoven, The Netherlands), ACM, 2010, pp. 127–136.

[28] C. Saito, A. Igarashi, and M. Viroli, *Lightweight Family Polymorphism*, JFP **18** (2008), no. 3, 285–331.

[29] I. Sommerville, *Software Engineering*, 9^{th} ed., Addison Wesley, 2011.

[30] W. Swierstra, *Data Types à la Carte*, JFP **18** (2008), no. 4, 423–436.

[31] C. Szyperski, *Independently Extensible Systems – Software Engineering Potential and Challenges*, Proc. 19^{th} Australasian Comp. Sci. Conf., 1996.

[32] M. Torgersen, *The Expression Problem Revisited*, Proc. 18^{th} ECOOP (Oslo (Norway)) (M. Odersky, ed.), LNCS, vol. 3086, June 2004, pp. 123–143.

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