Compositional Programming

Weixin Zhang, Yaozhu Sun, and Bruno C. d. S. Oliveira

Department of Computer Science
University of Hong Kong

4 May 2021

CP code is updated in favor of Zord

Expression Problem [Wad98]

The **expression problem** (EP) is a long-standing extensibility dilemma in programming languages.

From: wadler@research.bell-labs.com

The expression problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g. no casts). [.....]

Expression Problem (Code)

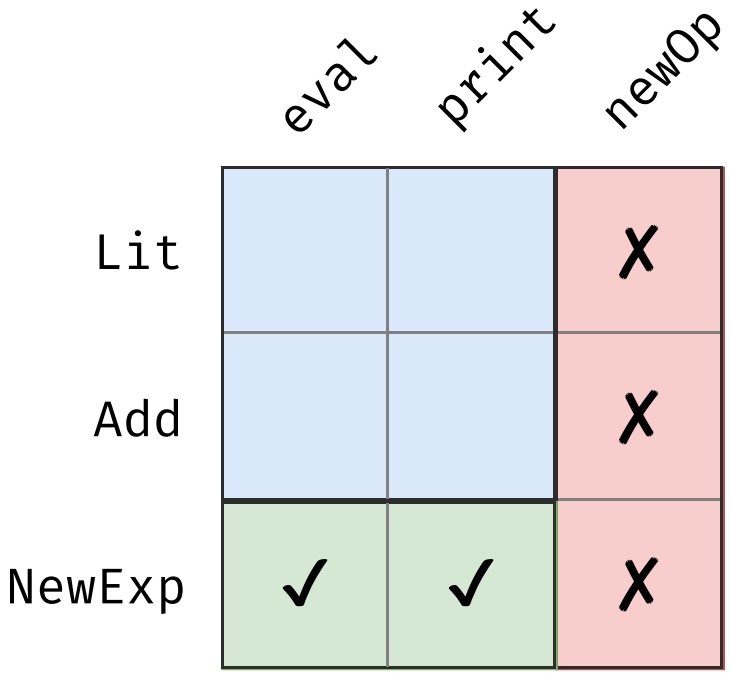
```scala
abstract class Exp {
  def eval:  Int
  def print: String
}
class Lit(n: Int) extends Exp {
  def eval  = n
  def print = n.toString
}
class Add(l: Exp, r: Exp)
    extends Exp {
  def eval  = l.eval + r.eval
  def print =
    l.print + "+" + r.print
}
```

OOP (Scala)

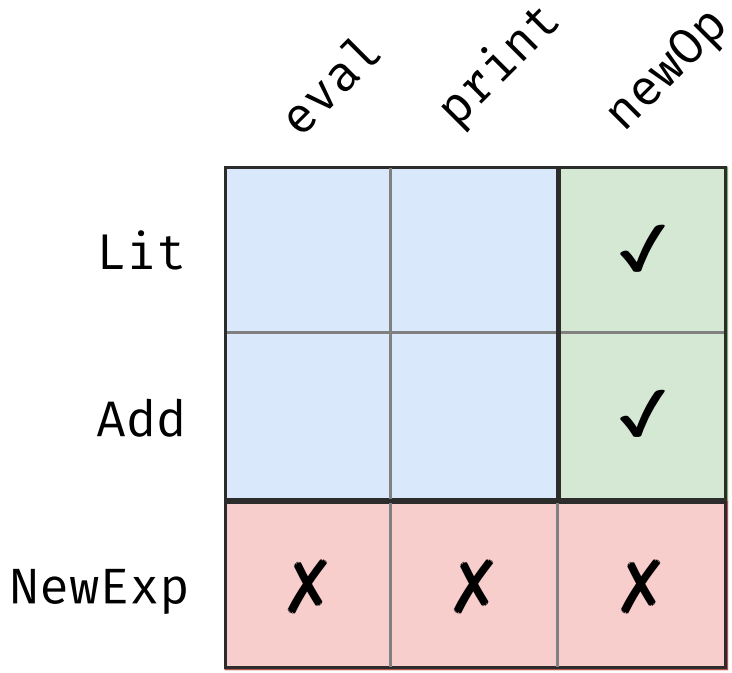
```haskell
data Exp where
  Lit :: Int -> Exp
  Add :: Exp -> Exp -> Exp

eval :: Exp -> Int
eval (Lit   n) = n
eval (Add l r) = eval l + eval r

print :: Exp -> String
print (Lit   n) = show n
print (Add l r) =
  print l ++ "+" ++ print r
```

FP (Haskell)

Expression Problem (Diagram)



OOP



FP

Requirements

There are five well-received requirements [ZO05] for solutions to the expression problem:

1. Extensibility in both dimensions
2. Strong static type safety
3. No modification or duplication
4. Separate compilation
5. Independent extensibility

Existing Solutions

There have been many approaches proposed to solve the expression problem. The most well-known solution in mainstream languages is *Church encodings*, also known as:

- *Tagless final* [OHL06, CKS07] in Haskell, and
- *Object algebras* [OC12] in OOP.

Tagless Final

```
class Exp e where
  lit :: Int -> e
  add :: e -> e -> e

......
```

Object Algebra

```
trait Exp[E] {
  def Lit(n: Int): E
  def Add(l: E, r: E): E
}
......
```

Limitations

However, existing solutions still have limitations. Take object algebras as an example:

- There is no proper *composition* mechanism for object algebras.
- It is hard to model *dependent* operations modularly.
- The programming style required to program with object algebras is quite unconventional compared to standard OOP.

These limitations partly arise from the lack of programming language support.

Overview

Compositional programming is a new statically typed modular programming style.

It offers an alternative way to model data structures that differs from both algebraic datatypes in functional programming and conventional OOP class hierarchies.

It allows us to naturally solve challenges such as the expression problem, model attribute-grammar-like programs, and generally deal with modular programs with *complex dependencies*.

Compositional Interfaces

```
type ExpSig<Exp> = {
  Lit : Int -> Exp;
  Add : Exp -> Exp -> Exp;
};
```

Compositional interfaces extend traditional OOP interfaces with *constructor signatures*. This enables programming the construction of objects against an interface, instead of a concrete implementation.

The type parameter `Exp` is called the *sort* of the compositional interface, which abstract over concrete types of objects. The return type of a constructor must always be a sort, which will be handled differently from normal type parameters.

Compositional Traits

```
type Eval = { eval : Int };

evalNum = trait implements ExpSig<Eval> => {
  (Lit   n).eval = n;
  (Add l r).eval = l.eval + r.eval;
}
```

Compositional traits extend typed first-class traits [BO18] with *virtual constructors*.

The outer trait `evalNum` is called a *trait family*, whose terminology is borrowed from family polymorphism [Ern01]. To put it simply, a trait family is a trait that contains nested virtual traits.

Method Patterns

```
-- Before:
evalNum = trait implements ExpSig<Eval> => {
  (Lit   n).eval = n;
  (Add l r).eval = l.eval + r.eval;
}
-- After:
evalNum = trait implements ExpSig<Eval> => {
  Lit   n = trait [self:Eval] => { eval = n };
  Add l r = trait [self:Eval] => { eval = l.eval + r.eval };
};
```

Method patterns provide a lightweight syntax for *nested traits*.

This enables compact method definitions for trait families, which resembles pattern matching in functional programming languages and attribute grammars.

Virtual Constructors & Self-Type Annotations

```
expAdd Exp = trait [self:ExpSig<Exp>] => {
  exp = new Add (new Lit 4) (new Lit 8);
};
```

`Add` and `Lit` (`self.` can be omitted) are both **virtual constructors**, which confirms to the signature in `ExpSig<Exp>`, rather than being bound to a specific implementation of a trait.

Self-type annotations provide a modular way to inject dependencies on other operations or constructors. Here, it implies that `expAdd` must be merged with some trait that concretely implements `ExpSig<Exp>` for instantiation later.

Extensibility: Adding a New Operation

```
type Print = { print : String };

printNum = trait implements ExpSig<Print> => {
  (Lit   n).print = toString n;
  (Add l r).print = "(" ++ l.print ++ "+" ++ r.print ++ ")";
};
```

Like in functional programming with pattern matching, adding a new operation is trivial: just to create an independent trait family that implements the compositional interface.

Nested Trait Composition

```
e = new evalNum , printNum , expAdd @(Eval&Print);
e.exp.print ++ " is " ++ toString e.exp.eval
--> "(4+8) is 12"
```

Nested trait composition is the mechanism used to compose compositional traits. It plays a similar role to traditional class inheritance, but generalizes to the dynamic composition of nested traits. It enables a form of inheritance of whole hierarchies, similar to the forms of composition found in family polymorphism.

Nested composition is performed by the *merge operator* , [Dun12]. At the type level, nested composition is supported by the BCD-style distributive subtyping [BCDC83].

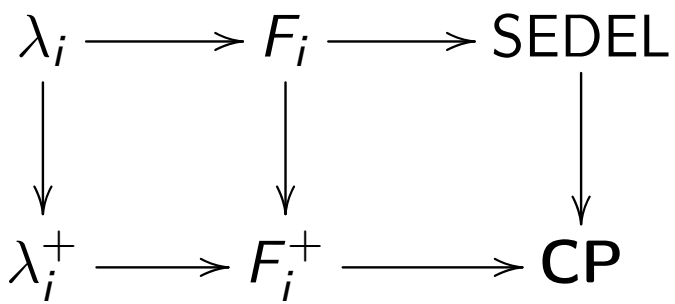
Extensibility: Adding a New Variant

```
type MulSig<Exp> = ExpSig<Exp> & {
  Mul : Exp -> Exp -> Exp;
};
evalMul = trait implements MulSig<Eval> inherits evalNum => {
  (Mul l r).eval = l.eval * r.eval;
};
printMul = trait implements MulSig<Print> inherits printNum => {
  (Mul l r).print = "(" ++ l.print ++ "*" ++ r.print ++ ")";
};

expMul Exp = trait [self:MulSig<Exp>] inherits expAdd @Exp => {
  override exp = new Mul super.exp (new Lit 4);
};
e' = new evalMul , printMul , expMul @(Eval&Print);
e'.exp.print ++ " is " ++ toString e'.exp.eval
--> "((4+8)*4) is 48"
```

The expression problem is solved!

Whence CP Originates?

ICFP'16, ESOP'17, ECOOP'18, ESOP'19, and TOPLAS'21



- λ_i Disjoint Intersection Types [OSA16]
- F_i Disjoint Polymorphism [AOS17]
- SEDEL Typed First-Class Traits [BO18]
- λ_i^+ The Essence of Nested Composition [BOS18]
- F_i^+ Distributive Disjoint Polymorphism for Compositional Programming [BXOS19]

Why Disjointness Matters?

Traits vs. Mixins

Disjointness brings *coherence* to calculi with intersection types and a merge operator. For example, `{ x = 1 } , { x = 2 }` is forbidden to avoid ambiguous semantics. The disjointness rule on record types naturally leads to *traits* (in contrast to *mixins*).

Traits	*explicitly* resolve conflicts; composition is *associative* and *commutative*.
Mixins	*implicitly* resolve conflicts; composition order affects semantics.

Therefore, the trait model avoids unexpected errors caused by the wrong choice of implementation through implicit resolution.

Weaker Dependencies for Stronger Modularity

In the original expression problem, there are very few dependencies. Matters become significantly more complicated in the presence of more advanced forms of dependencies, e.g. modeling **attribute grammars** (AG).

In compositional programming, there are two primary mechanisms to deal with dependencies:

- Compositional interface type refinement (input part of sorts);
- Self-type annotations (family & object self-references).

Dependencies: Synthesized Attributes

```
-- Child dependencies:
printChild = trait implements ExpSig<Eval => Print> => {
  (Lit   n).print = toString n;
  (Add l r).print = if r.eval == 0 then l.print
      else "(" ++ l.print ++ "+" ++ r.print ++ ")";
};
-- Self dependencies:
printSelf = trait implements ExpSig<Eval => Print> => {
  (Lit   n).print = toString n;
  (Add l r).print = if self.eval == 0 then "0"
      else "(" ++ l.print ++ "+" ++ r.print ++ ")";
};
```

In both examples, the sort types are refined in input positions. The synthesized attribute `print` weakly depends on `eval`, whose implementation is modularly defined elsewhere.

Object self-references `[self:Eval]` are implicitly added.

Dependencies: Inherited Attributes

```
-- Initially:
type Eval = { eval : Int };
evalNum = trait implements ExpSig<Eval> => {
  (Lit   n).eval = n;
  (Add l r).eval = l.eval + r.eval;
};
-- After variable binding is introduced:
type Eval = { eval : EnvN -> Int };
evalNum = trait implements ExpSig<Eval> => {
  (Lit   n).eval  _   = n;
  (Add l r).eval env = l.eval env + r.eval env;
};
-- After function namespace is introduced:
type Eval = { eval : EnvN -> EnvF -> Int };
evalNum = trait implements ExpSig<Eval> => {
  (Lit   n).eval   _     _   = n;
  (Add l r).eval envN envF = l.eval envN envF + r.eval envN
      envF;
};
```

Polymorphic Contexts

Compositional programming interacts with (disjoint) parametric polymorphism: `Eval` serves as a *Reader* monad.

```
type Eval Context = { eval : Context -> Int };

-- Initial planning for polymorphic contexts:
evalNum Context = trait implements ExpSig<Eval Context> => {
  (Lit   n).eval  _   = n;
  (Add l r).eval ctx = l.eval ctx + r.eval ctx;
};

-- Smoothly adding variables and binders:
type CtxN = { envN : EnvN };
evalVar (Context * CtxN) =
    trait implements VarSig<Eval (CtxN&Context)> => {
  -- polymorphic record update
  (Let s e e').eval ctx = e'.eval ({ envN = insert @Int s
      (e.eval ctx) ctx.envN } , (ctx:Context));
  (Var     s).eval ctx = lookup @Int s ctx.envN;
};
```

Polymorphic Contexts (cont.)

```
-- Smoothly adding function definitions and applications:
type CtxF = { envF : EnvF };
evalFunc (Context * CtxF) =
    trait implements FuncSig<Eval (CtxF&Context)> => {
  (LetF s f e).eval ctx = e.eval ({ envF = insert @Func s f
      ctx.envF } , (ctx:Context));
  (AppF  s  e).eval ctx = (lookup @Func s ctx.envF) (e.eval
      ctx);
};

expPoly E = trait [self : ExpSig<E>&VarSig<E>&FuncSig<E>] => {
  -- let f = \x -> x * x in let x = 9 in f x
  exp = new LetF "f" (\(x:Int) -> x * x)
      (new Let "x" (new Lit 9) (new AppF "f" (new Var "x")));
};

-- Context types used by other trait families are passed:
e = new evalNum @(CtxN&CtxF) , evalVar @CtxF , evalFunc @CtxN ,
        expPoly @(Eval (CtxN&CtxF));
e.exp.eval { envN = empty @Int; envF = empty @Func }  --> 81
```

Polymorphic Contexts (cont.)

Compositional programming with polymorphic contexts can also model other forms of attribute grammars:

- *L-attributed grammars*, where inherited attributes depend on both parents and *left siblings*;
- *Chained attributes*, which are both inherited and synthesized, like *State* monads.

Polymorphic contexts provide us with a simple approach to modularly handle new contexts without changing the existing code, while keeping strong static type safety.

The significant advantage is that the polymorphic portion of the context cannot be fiddled with, i.e. the only thing one can do is to pass it intact. Nevertheless, polymorphic contexts can still be refined for particular uses and expose just the right amount of information while hiding the remaining information.

Syntax

Program	P	$::=$	$D; P \mid E$
Declarations	D	$::=$	$M \mid \textbf{type}\ X\langle\overline{\alpha}\rangle = A$
Term decls	M	$::=$	$x = E \mid (L\ \overline{(x : A)}\ [\textbf{self} : B]).I = E$
Types	A, B	$::=$	$\textbf{Int} \mid \alpha \mid \top \mid \bot \mid A \to B \mid \forall(\alpha * A).B$
		$\mid$	$A\ \&\ B \mid \{I : A\} \mid \textbf{Trait}\langle A \Rightarrow B\rangle \mid X\overline{S}$
Sorts	S	$::=$	$\langle A\rangle \mid \langle A \Rightarrow B\rangle$
Expressions	E	$::=$	$i \mid x \mid () \mid \lambda x.E \mid E_1\ E_2 \mid \Lambda(\alpha * A).E$
		$\mid$	$E\ @A \mid E_1\ ,\ E_2 \mid \{M\} \mid E.I \mid E : A \mid E_1\ \hat{}\ E_2$
		$\mid$	$\textbf{let}\ x = E_1\ \textbf{in}\ E_2 \mid \textbf{open}\ E_1\ \textbf{in}\ E_2 \mid \textbf{new}\ E$
		$\mid$	$\textbf{trait}[\textbf{self} : A]\ \textbf{implements}\ B\ \textbf{inherits}\ E_1 \Rightarrow E_2$
Term context	Γ	$::=$	$\bullet \mid \Gamma, x : A$
Type context	Δ	$::=$	$\bullet \mid \Delta, \alpha * A \mid \Delta, X\overline{S} \mapsto A$
Sort context	Σ	$::=$	$\bullet \mid \Sigma, \alpha \mapsto \beta$

Semantics

The semantics is given by type-directed elaboration to a call-by-name variant of the F_i^+ calculus [BXOS19] extended with (recursive) let bindings.

During the elaboration, some transformations and checks are performed:

- Type declarations are transformed to distinguish the input/output occurrences of sorts;
- Type synonyms and sort instantiations are expanded;
- Disjointness and subtyping relations are checked, both of which rely on the concept of top-like types.

The elaboration builds on two basic ideas: generalized object algebras [OSLC13] and a well-known denotational model of inheritance [CP89].

Sort Transformation & Type Expansion

Before desugaring

```
type ExpSig<Exp> = {
  Lit : Int -> Exp;
  Add : Exp -> Exp -> Exp;
};

type Sig = ExpSig<Eval => Print>;
```

After desugaring

```
type ExpSig Exp OExp =
  { Lit : Int -> Trait<Exp => OExp> } &
  { Add : Exp -> Exp -> Trait<Exp => OExp> };

type Sig = ExpSig (Eval&Print) Print;
```

Selected Typing Rules

$$\boxed{\Delta; \Gamma \vdash E \Rightarrow A \rightsquigarrow e}$$

T-trait

$$\frac{\begin{array}{c}\Delta; \bullet \vdash A \Rightarrow A_1 \\ \Delta; \bullet \vdash B \Rightarrow B_1 \qquad \Delta; \Gamma, \mathbf{self} : A_1 \vdash E_1 \Rightarrow \mathbf{Trait}\langle A_2 \Rightarrow B_2 \rangle \rightsquigarrow e_1 \qquad A_1 <: A_2 \\ \Delta; \Gamma, \mathbf{self} : A_1, \mathbf{super} : B_2 \vdash E_2 \Rightarrow C \rightsquigarrow e_2 \qquad C * B_2 \qquad C \,\&\, B_2 <: B_1\end{array}}{\begin{array}{l}\Delta; \Gamma \vdash \mathbf{trait}[\mathbf{self} : A]\ \mathbf{implements}\ B\ \mathbf{inherits}\ E_1 \Rightarrow E_2 \Rightarrow \mathbf{Trait}\langle A_1 \Rightarrow C \,\&\, B_2 \rangle \\ \quad \rightsquigarrow \lambda(\mathbf{self} : |A_1|).\ \mathbf{let\ super} = e_1\ \mathbf{self\ in}\ e_2\ ,\ \mathbf{super}\end{array}}$$

T-mergeTrait

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash E_1 \Rightarrow \mathbf{Trait}\langle A_1 \Rightarrow B_1 \rangle \rightsquigarrow e_1 \\ \Delta; \Gamma \vdash E_2 \Rightarrow \mathbf{Trait}\langle A_2 \Rightarrow B_2 \rangle \rightsquigarrow e_2 \qquad \Delta \vdash B_1 * B_2\end{array}}{\Delta; \Gamma \vdash E_1\ ,\ E_2 \Rightarrow \mathbf{Trait}\langle A_1 \,\&\, A_2 \Rightarrow B_1 \,\&\, B_2 \rangle \rightsquigarrow \lambda(\mathbf{self} : |A_1 \,\&\, A_2|).\ e_1\ \mathbf{self}\ ,\ e_2\ \mathbf{self}}$$

T-new

$$\frac{\Delta; \Gamma \vdash E \Rightarrow \mathbf{Trait}\langle A \Rightarrow B \rangle \rightsquigarrow e \qquad B <: A}{\Delta; \Gamma \vdash \mathbf{new}\ E \Rightarrow B \rightsquigarrow \mathbf{let\ self} : |B| = e\ \mathbf{self\ in\ self}}$$

Selected Subtyping Rules

$\boxed{A <: B}$

S-topTrait

$$\top <: \mathbf{Trait}\langle \top \Rightarrow \top \rangle$$

S-trait

$$\frac{A_2 <: A_1 \qquad B_1 <: B_2}{\mathbf{Trait}\langle A_1 \Rightarrow B_1 \rangle <: \mathbf{Trait}\langle A_2 \Rightarrow B_2 \rangle}$$

S-distTrait

$$\mathbf{Trait}\langle A \Rightarrow B \rangle \,\&\, \mathbf{Trait}\langle A \Rightarrow C \rangle <: \mathbf{Trait}\langle A \Rightarrow B \,\&\, C \rangle$$

Metatheory

Theorem (Type-safety)

If $\Delta; \Gamma \vdash P \Rightarrow A \rightsquigarrow e$ *then* $|\Delta|; |\Gamma| \vdash e \Rightarrow |A|$.

Proof.

By structural induction on the typing judgment. □

Theorem (Coherence)

Every well-typed CP program has a unique elaboration.

Proof.

For every elaboration rule, the elaborated F_i^+ expression in the conclusion is uniquely determined by the elaborated F_i^+ expressions in the premises. By the coherence property of F_i^+, we conclude that each well-typed CP program has a unique elaboration. □

Conclusion

We presented four key concepts of compositional programming: compositional interfaces, compositional traits, method patterns, and nested trait composition, as well as its ability to solve the expression problem and handle non-trivial dependencies. Moreover, the calculus of CP is proved to be type-safe and coherent.

Our artifact contains a Haskell implementation of CP. There are also three case studies:

- *Scans*, a DSL for describing parallel prefix circuits;
- A *mini interpreter* for an expression language;
- *C0 compiler*, which compiles a subset of C to JVM instructions.

Future Work

There is still room to make compositional programming more expressive and practical, e.g.:

- to integrate *recursive types* and *type constructors*;
- to model imperative objects with *mutable references*;
- to better support *type inference* [Ber19], esp. to infer type arguments in parametric polymorphism.

References

[AOS17] João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi, *Disjoint polymorphism*, ESOP, 2017.

[BCDC83] Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini, *A filter lambda model and the completeness of type assignment*, JSL **48** (1983), no. 4.

[Ber19] Birthe van den Berg, *Type inference for disjoint intersection types*, Master's thesis, KU Leuven, 2019.

[BO18] Xuan Bi and Bruno C. d. S. Oliveira, *Typed first-class traits*, ECOOP, 2018.

[BOS18] Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers, *The essence of nested composition*, ECOOP, 2018.

[BXOS19] Xuan Bi, Ningning Xie, Bruno C. d. S. Oliveira, and Tom Schrijvers, *Distributive disjoint polymorphism for compositional programming*, ESOP, 2019.

[CKS07] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan, *Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages*, APLAS, 2007.

References (cont.)

[CP89] William Cook and Jens Palsberg, *A denotational semantics of inheritance and its correctness*, OOPSLA, 1989.

[Dun12] Jana Dunfield, *Elaborating intersection and union types*, ICFP, 2012.

[Ern01] Erik Ernst, *Family polymorphism*, ECOOP, 2001.

[OC12] Bruno C. d. S. Oliveira and William R. Cook, *Extensibility for the masses: Practical extensibility with object algebras*, ECOOP, 2012.

[OHL06] Bruno C. d. S. Oliveira, Ralf Hinze, and Andres Löh, *Extensible and modular generics for the masses*, TFP, 2006.

[OSA16] Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim, *Disjoint intersection types*, ICFP, 2016.

[OSLC13] Bruno C. d. S. Oliveira, Tijs van der Storm, Alex Loh, and William R. Cook, *Feature-oriented programming with object algebras*, ECOOP, 2013.

[Wad98] Philip Wadler, *The expression problem*, Note to Java Genericity mailing list, 1998.

[ZO05] Matthias Zenger and Martin Odersky, *Independently extensible solutions to the expression problem*, FOOL@POPL, 2005.