The expression lemma*

Ralf Lämmel[1] and Ondrej Rypacek[2]

[1] The University of Koblenz-Landau, Germany
[2] The University of Nottingham, UK

Abstract. Algebraic data types and catamorphisms (folds) play a central role in functional programming as they allow programmers to define recursive data structures and operations on them uniformly by structural recursion. Likewise, in object-oriented (OO) programming, recursive hierarchies of object types with virtual methods play a central role for the same reason. There is a semantical correspondence between these two situations which we reveal and formalize categorically. To this end, we assume a coalgebraic model of OO programming with functional objects. The development may be helpful in deriving refactorings that turn sufficiently disciplined functional programs into OO programs of a designated shape and vice versa.

Key words: expression lemma, expression problem, functional object, catamorphism, fold, the composite design pattern, program calculation, distributive law, free monad, cofree comonad.

1 Introduction

There is a folk theorem that goes as follows. Given is a recursively defined data structure with variants $d_1, \ldots, d_q$, and operations $o_1, \ldots, o_r$ that are defined by structural recursion on the data variants. There are two equivalent implementations. In the *functional* style, we define recursive functions $f_1, \ldots, f_r$ such that each f_i implements o_i and is defined by q equations, one equation for each d_j. In the *object-oriented* (OO) style, we define object types $t_1, \ldots, t_q$ such that each t_j uses d_j as its opaque state type, and it implements a common interface consisting of methods $m_1, \ldots, m_r$ with the types of $f_1, \ldots, f_r$, except that the position of structural recursion is mapped to "self" (say, "this"). The per-variant equations of the functional style correspond to the per-variant method implementations of the OO style. Refer to Fig. 1 for a Haskell and a Java program that are related in the described manner.

This folk theorem is related to the so-called *expression problem* [28], which focuses on the extensibility trade-offs of the different programming styles, and aims at improved language designs with the best possible extensibility for all possible scenarios. Such a comparison of styles can definitely benefit from an understanding of the semantical correspondence between the styles, which is indeed the overall contribution of the present paper. We coin the term *expression lemma* to refer to the sketched functional/OO correspondence. We do not discuss the expression problem any further in this paper, but

* See `http://www.uni-koblenz.de/~laemmel/expression/` (the paper's web site) for an extended version.

```
-- Arithmetic expression forms
data Expr = Num Int |Add Expr Expr

-- Evaluate expressions
eval :: Expr → Int
eval (Num i) = i
eval (Add l r) = eval l + eval r

-- Modify literals modulo v
modn :: Expr → Int → Expr
modn (Num i) v = Num (i `mod` v)
modn (Add l r) v = Add (modn l v) (modn r v)
```

```
public abstract class Expr {
    public abstract int eval();
    public abstract void modn(int v);
}

public class Num extends Expr {
    private int value;
    public Num(int value) { this.value = value; }
    public int eval() { return value; }
    public void modn(int v) { this.value = this.value % v; }
}

public class Add extends Expr {
    private Expr left, right;
    public Add(Expr left, Expr right) { this.left = left; this.right = right; }
    public int eval() { return left.eval() + right.eval(); }
    public void modn(int v) { left.modn(v); right.modn(v); }
}
```

Fig. 1. A Haskell program and a Java program; the two programs define the same kind of structurally recursive operations on the same kind of recursive data structure.

we contend that the expression lemma backs up past and future work on the expression problem. It is also assumed that the lemma contributes to the foundation that is needed for future work on refactorings between the aforementioned styles, e.g., in the context of making imperative OO programs more pure, more functional, or more parallelizable.

Contributions

The paper provides a technical formulation of the expression lemma, in fact, the first such formulation, as far as we know. The formulation is based on comparing functional programs (in particular, folds) with coalgebraically modeled OO programs. We provide first ever, be it partial answers to the following questions:

- When is a functional program that is defined by recursive functions on an algebraic data type semantically equivalent to an OO program that is defined by recursive methods on object structures?
- What is the underlying common definition of the two programs?

The formal development is done categorically, and essentially relies on the established concept of distributive laws of a functor over a functor and (co)monadic generalizations

thereof. A class of pairs of functional and OO programs that are duals of each other is thus revealed and formalized. This makes an important contribution to the formal understanding of the semantical equivalence of functional and OO programming. Nontrivial facets of dualizable programs are identified — including facets for the freewheeling use of term construction (say, object construction) and recursive function application (say, method invocation).[1]

Road-map

The paper is organized as follows. § 2 sketches (a simple version of) the expression lemma and its proof. § 3 approaches the expression lemma categorically. § 4 interprets the basic formal development, and suggests extensions to cover a larger class of dualizable programs. § 5 formalizes the proposed extensions. § 6 discusses related work. § 7 concludes the paper.

2 Informal development

In order to clarify the correspondence between the functional and the OO style, we need a setup that admits the comparison of both kinds of programs. In particular, we must introduce a suitable formalism for objects. We adopt the coalgebraic view of functional objects, where object interfaces are modeled as *interface endofunctors*, and implementations are modeled as *coalgebras* of these functors. We will explain the essential concepts here, but refer to [22,14] for a proper introduction to the subject, and to [21] for a type-theoretical view. In the present section, we illustrate the coalgebraic model in Haskell, while the next section adopts a more rigorous, categorical approach.

2.1 Interfaces and coalgebras

The following type constructor models the *interface* of the base class Expr of Fig. 1:

```
type IExprF x = (Int, Int → x)
```

Here, x is the type parameter for the object type that implements the interface; the first projection corresponds to the observer eval (and hence its type is Int); the second projection corresponds to the modifier modn (and hence its type is Int→x, i.e., the method takes an Int and returns a new object of the same type x). We also need a type that hides the precise object type — in particular, its state representation; this type would effectively be used as a bound for all objects that implement the interface. To this end, we may use the recursive closure of IExprF:

```
newtype IExpr = InIExpr { outIExpr :: IExprF IExpr }
```

Method calls boil down to the following convenience projections:

[1] The paper comes with a source distribution that contains a superset of all illustrations in the paper as well as a Haskell library that can be used to code dualizable programs. Refer to the paper's web site.

```
-- eval function for literals
numEval :: Int → Int
numEval = id
```

```
-- modn function for literals
numModn :: Int → Int
numModn = mod
```

```
-- eval function for additions
addEval :: (Int, Int) → Int
addEval = uncurry (+)
```

```
-- modn function for addition
addModn :: (Int→a,Int→a)→Int→(a,a)
addModn = uncurry (/\)
```

Fig. 2. Component functions of the motivating example.

```
call_eval = fst . outIExpr
call_modn = snd . outIExpr
```

Implementations of the interface are *coalgebras* of the IExprF functor. Thus, the type of IExprF implementations is the following:

```
type IExprCoalg x = x → IExprF x
```

Here are the straightforward implementations for the Expr hierarchy:

```
numCoalg :: IExprCoalg Int
numCoalg = numEval /\ numModn

addCoalg :: IExprCoalg (IExpr, IExpr)
addCoalg = (addEval . (call_eval <*> call_eval)) /\
           (addModn . (call_modn <*> call_modn))
```

For clarity (and reuse), we have factored out the actual functionality per function and data variant into helper functions numEval, addEval, numModn, and addModn; c.f. Fig. 2.[2] Note that the remaining polymorphism of addModn is essential to maintain enough naturality needed for our construction.

The above types clarify that the implementation for literals uses int as the *state type*, whereas the implementation for additions uses (IExpr, IExpr) as the state type. Just as the original Java code invoked methods on the components stored in left and right, the coalgebraic code applies the corresponding projections of the IExpr-typed values call_eval and call_modn. It is important to keep in mind that IExpr-typed values are effectively functional objects, i.e., they return a "modified (copy of) self", when mutations are to be modeled.

[2] We use point-free (pointless) notation (also known as Backus' functional forms) throughout the paper to ease the categorical development. In particular, we use these folklore operations: f <*> g maps the two argument functions over the components of a pair; f /\ g constructs a pair by applying the two argument functions to the same input; f <|> g maps over a sum with an argument function for each case; f \/ g performs case discrimination on a sum with an argument function for each case. Refer to Fig. 3 for a summary of the operations.

```
(<*>) :: (a → b) → (c → d) → (a,c) → (b,d)
(f <*> g) (x, y) = (f x, g y)

(/\) :: (a → b) → (a → c) → a → (b,c)
(f /\ g) x = (f x, g x)

(<|>) :: (a → b) → (c → d) → Either a c → Either b d
(f <|> g) (Left x)  = Left (f x)
(f <|> g) (Right y) = Right (g y)

(\/) :: (a → c) → (b → c) → Either a b → c
(f \/ g) (Left x)  = f x
(f \/ g) (Right y) = g y
```

Fig. 3. Folklore operations on sums and products.

2.2 Object construction by unfolding

Given a coalgebra $x \longrightarrow f\ x$ for some fixed f and x, we can *unfold* a value of type x to the type of the fixed point of f. The corresponding well-known recursion scheme of *anamorphisms* [19] is instantiated for IExpr as follows:

```
unfoldIExpr :: IExprCoalg x → x → IExpr
unfoldIExpr c = InIExpr . fmapIExprF (unfoldIExpr c) . c
  where
    fmapIExprF :: (x → y) → IExprF x → IExprF y
    fmapIExprF f = id <*> (.) f
```

That is, IExprF x is injected into IExpr by recursively unfolding all occurrences of x by means of the functorial map operation, fmapIExprF. The type parameter x occurs in the positions where "a modified (copy of) self" is returned. In OO terms, applications of unfoldIExpr are to be viewed as *constructor* methods:

```
newNum :: Int → IExpr
newNum = unfoldIExpr numCoalg

newAdd :: (IExpr, IExpr) → IExpr
newAdd = unfoldIExpr addCoalg
```

This completes the transcription of the Java code of Fig. 1 to the coalgebraic setup.

2.3 Converting state trees to object trees

In establishing a semantical correspondence between functional and OO programs, we can exploit the following property: *the functional style is based on recursion into terms whose structure coincides with the states of the objects.* Hence, let us try to convert such trees of states of objects (state trees) into trees of objects (object trees). We will then need to compare the semantics of the resulting objects with the semantics of the functional program.

In the introduction, we defined the data type Expr as an algebraic data type; for the sake of a more basic notation, we define it here as the fixed point of a "sum-of-products"

functor ExprF equipped with convenience injections num and add — reminiscent of the algebraic data-type constructors of the richer notation. Thus:

```
type ExprF x = Either Int (x,x)
newtype Expr = InExpr { outExpr :: ExprF Expr }
num = InExpr . Left
add = InExpr . Right
```

The objects we construct have internal state either of type Int or of type (IExpr,IExpr). The corresponding sum, Either Int (IExpr,IExpr), coincides with ExprF IExpr, i.e., the mere state trees resemble the term structure in the functional program. We have defined coalgebras numCoalg and addCoalg for each type of state. We can also define a coalgebra for the union type ExprF IExpr:

```
eitherCoalg :: IExprCoalg (ExprF IExpr)
eitherCoalg = ((id <*> (.) Left) \/ (id <*> (.) Right)) .
              (numCoalg <|> addCoalg)
```

The coalgebra eitherCoalg may be viewed as an implementation of an object type that physically uses a sum of the earlier state types. Alternatively, we may view the coalgebra as an object factory (in the sense of the *abstract factory* design pattern [10]). That is, we may use it as a means to construct objects of either type.

```
newEither :: ExprF IExpr → IExpr
newEither = unfoldIExpr eitherCoalg
```

It is important to understand the meaning of ExprF IExpr: the type describes states of objects, where the state representation is only exposed at the top-level, but all deeper objects are already opaque and properly annotated with behavior. While newEither facilitates one level of object construction, we ultimately seek the recursive closure of this concept. That is, we seek to convert a pure state tree to a proper object tree. It turns out that the fold operation for Expr immediately serves this purpose.

In general, the fold operation for a given sums-of-products functor is parametrized by an *algebra* that associates each addend of the functor's sum with a function that combines recursively processed components and other components. For instance, the algebra type for expressions is the following:

```
type ExprAlg x = ExprF x →x
```

Given an algebra $f\ x \longrightarrow x$ for some fixed f and x, we can *fold* a value of the type of the fixed point of f to a value of type x. The corresponding well-known recursion scheme of *catamorphisms* is instantiated for Expr as follows:

```
foldExpr :: ExprAlg x → Expr → x
foldExpr a = a . fmapExprF (foldExpr a) . outExpr
  where
    fmapExprF :: (x → y) → ExprF x → ExprF y
    fmapExprF f = id <|> (f <*> f)
```

We should simplify the earlier type for newEither as follows:

```
newEither :: ExprAlg IExpr
```

Hence, we can fold over state trees to obtain object trees.

```
-- Fold the unfold
fu :: Expr → IExpr
fu = foldExpr newEither
```

2.4 Implementing interfaces by folds over state trees

It remains to compare the semantics of the constructed objects with the semantics of the corresponding functional program. To this end, we also model the construction of objects whose object type corresponds to an abstract data type (ADT) that exports the functions of the functional program as its operations.

The initial definitions of the functions eval and modn used general recursion. Our development relies on the fact that the functions are catamorphisms (subject to further restrictions). Here are the new definitions; subject to certain preconditions, programs that use general recursion can be automatically converted to programs that use the catamorphic scheme [17,11]:

```
evalAlg :: ExprAlg Int
evalAlg = numEval \/ addEval

eval :: Expr → Int
eval = foldExpr evalAlg

modnAlg :: ExprAlg (Int → Expr)
modnAlg = ((.) num . numModn) \/ ((.) add . addModn)

modn :: Expr → Int → Expr
modn = foldExpr modnAlg
```

Just like objects combine behavior for all operations in their interfaces, we may want to tuple the folds, such that all recursions are performed simultaneously. That is, the result type of the paired fold is the product of the result types of the separated folds. Again such pairing (tupling) is a well-understood technique [9,19,5,12] that can also be used in an automated transformation. Thus:

```
bothAlg :: ExprAlg (IExprF Expr)
bothAlg = (evalAlg <*> modnAlg) . ((id <|> (fst <*> fst)) /\
            (id <|> (snd <*> snd)))

both :: IExprCoalg Expr
both = foldExpr bothAlg
```

That is, both does both, eval and modn. Now we can construct objects whose behavior is immediately defined in terms of both (hence, essentially, in terms of the original functions eval and modn). To this end, it is sufficient to realize that both readily fits as a coalgebra, as evident from its type:

```
Expr → (Int, Int → Expr) ≡ Expr → IExprF Expr ≡ IExprCoalg Expr
```

Thus, we can construct objects (ADTs, in fact) as follows:

```
-- Unfold the fold
uf :: Expr → IExpr
uf = unfoldIExpr both
```

That is, we have encapsulated the functional folds with an argument term such that the resulting interface admits the applications of the functional folds to the term.

2.5 The expression lemma

Let us assume that we were able to prove the following identity:

```
foldExpr (unfoldIExpr eitherCoalg) = unfoldIExpr (foldExpr bothAlg)
```

We refer to the generic form of this identity as the expression lemma; c.f. § 3.4. Roughly, the claim means that *objects that were constructed from plain state trees, level by level, behave the same as shallow objects that act as abstract data types with the functional folds as operations*. Hence, this would define a proper correspondence between anamorphically (and coalgebraically) phrased OO programs and catamorphically phrased functional programs. § 3 formalizes this intuition and proves its validity.

The formal development will exploit a number of basic categorical tools, but a key insight is that the defining coalgebra of the OO program (i.e., eitherCoalg) and the defining algebra of the functional program (i.e., bothAlg) essentially involve the same function (in fact, a *natural transformation*). To see this, consider the expanded types of the (co)algebras in question:

```
eitherCoalg :: ExprF IExpr→ IExprF (ExprF IExpr)
bothAlg :: ExprF (IExprF Expr)→ IExprF Expr
```

The types differ in the sense that eitherCoalg uses the recursive closure IExpr in one position where bothAlg uses an application of the functor IExprF instead, and bothAlg uses the recursive closure Expr in another position where eitherCoalg uses an application of the functor ExprF instead. Assuming a natural transformation lambda, both functions can be defined as follows:

```
eitherCoalg = lambda . fmapExprF outIExpr
bothAlg = fmapIExprF InExpr . lambda
lambda :: ExprF (IExprF x) → IExprF (ExprF x)
lambda = ???
```

Note that naturality of lambda is essential here. It turns out that we can define lambda in a "disjunctive normal form" over the same ingredients that we also used in the original definitions of eitherCoalg and bothAlg:

```
lambda = (numEval /\ ((.) Left . numModn)) \/ ((addEval . (fst <*> fst)) /\
           ((.) Right . addModn . (snd <*> snd)))
```

It is straightforward to see that eitherCoalg and bothAlg as redefined above equate to the original definitions of eitherCoalg and bothAlg based on just trivial laws for sums and products. We have thus factored out a common core, a distributive law, lambda, from which both programs can be canonically defined.

3 The basic categorical model

The intuitions of the previous section will now be formalized categorically. The used categorical tools are established in the field of functional programming theory. The contribution of the section lies in leveraging these known tools for the expression lemma.

3.1 Interface functors

Definition 1. *An* interface functor *(or simply* interface*) is a polynomial endofunctor on a category* [3] $\mathbf{C}$ *of the form*

$$O \times M \;\; \textit{with} \;\; O = \prod_{i \in I} A_i^{B_i} \;\; \textit{and} \;\; M = \prod_{j \in J} (C_j \times \mathsf{Id})^{D_j}$$

where $\prod$ *denotes iterated product, all* As *,* Bs *,* Cs *and* Ds *are constant functors,* Id *is the identity functor and all products and exponents are lifted to functors.* I *and* J *are finite sets.*

Note that here and in the rest of the text we use the exponential notation for the function space as usual. Informally, O collects all "methods" that do not use the type of "self" in their results, as it is the case for observers; M collects all "methods" that return a "mutated (copy of) self" (c.f. the use of "Id"), and possibly additional data (c.f. the Cs).

Example 1. IExprF is an interface functor.

$$\mathsf{IExprF} \;=\; \mathsf{Int} \times \mathsf{Id}^{\mathsf{Int}} \;\cong\; \mathsf{Int}^1 \times (1 \times \mathsf{Id})^{\mathsf{Int}}$$

3.2 F-(co)algebras and their morphisms

Definition 2. *Let* F *be an endofunctor on a category* $\mathbf{C}$*,* A*,* B *objects in* $\mathbf{C}$*.*

- *An* F***-algebra*** *is an arrow* $F\,A \longrightarrow A$*. Here* A *is called the* ***carrier*** *of the algebra.*
- *For* F*-algebras* $\varphi : F\,A \longrightarrow A$ *and* $\psi : F\,B \longrightarrow B$*, an* F***-algebra morphism*** *from* φ *to* ψ *is an arrow* $f : A \longrightarrow B$ *of* $\mathbf{C}$ *such that the following holds:*

$$f \circ \varphi \;=\; \psi \circ Ff \tag{1}$$

- F*-algebras and* F*-algebra morphisms form a category denoted* $\mathbf{C}^F$*. The initial object in this category, if it exists, is the* ***initial*** F***-algebra****. Explicitly, it is an* F*-algebra,* $\mathsf{in}_F : F\,\mu F \longrightarrow \mu F$*, such that for any other* F*-algebra,* $\varphi : F\,A \longrightarrow A$*, there exists a* unique F*-algebra morphism* $(\!|\, \varphi \,|\!)_F$ *from* in_F *to* φ*. Equivalently:*

$$h \;=\; (\!|\, \varphi \,|\!)_F \;\Leftrightarrow\; h \circ \mathsf{in}_F \;=\; \varphi \circ Fh \tag{2}$$

[3] For simplicity, in this paper, we assume a category $\mathbf{C}$ with enough structure to support our constructions. The category $\mathbf{SET}$ is always a safe choice.

– *The duals of the above notions are* F***-coalgebra****,* F***-coalgebra morphism*** *and the* ***terminal*** F***-coalgebra****. Explicitly, an* F*-coalgebra is an arrow* $\varphi : A \longrightarrow FA$ *in* $\mathbf{C}$*. The category of* F*-coalgebras is denoted* $\mathbf{C}_F$*. The terminal* F*-coalgebra is denoted* $\mathsf{out}_F : \nu F \longrightarrow F\nu F$*, and the unique terminal* F*-coalgebra morphism from* φ *is denoted* $[\!(\, \varphi \,)\!]_F : A \longrightarrow \nu F$*. These satisfy the following duals of (1) and (2).*

$$\psi \circ g \;=\; Fg \circ \varphi \tag{3}$$

$$h \;=\; [\!(\, \varphi \,)\!]_F \Leftrightarrow \mathsf{out}_F \circ h \;=\; Fh \circ \varphi \tag{4}$$

Example 2. (Again, we relate to the Haskell declarations of § 2.) ExprF is an endofunctor; μExprF corresponds to type Expr; evalAlg and modnAlg are ExprF-algebras. The combinator $(\!|\ |\!)_{\mathsf{ExprF}}$ corresponds to foldExpr. Likewise, IExprF is an endofunctor; νIExprF corresponds to IExpr; numCoalg and addCoalg are IExprF-coalgebras. The combinator $[\!(\ )\!]_{\mathsf{IExprF}}$ corresponds to unfoldIExpr.

3.3 Simple distributive laws

Our approach to proving the correspondence between OO and functional programs critically relies on distributive laws as a means to relate algebras and coalgebras. In the present section, we only introduce the simplest form of distributive laws.

Definition 3. *A* ***distributive law*** *of a* ***functor*** F *over a* ***functor*** B *is a natural transformation* $FB \longrightarrow BF$*.*

Example 3. Trivial examples of distributive laws are algebras and coalgebras. That is, *any* coalgebra $X \longrightarrow BX$ is a distributive law where F in Def. 3 is fixed to be a constant functor. Dually, *any* algebra $FX \longrightarrow X$ is a distributive law where B in Def. 3 is fixed to be a constant functor. This fact is convenient in composing algebras and coalgebras (and distributive laws), as we will see shortly.

Example 4. Here are examples of nontrivial distributive laws:[4]

$$\mathsf{addModn} \;:\; \mathsf{Id}^2\,\mathsf{Id}^{\mathsf{Int}} \longrightarrow \mathsf{Id}^{\mathsf{Int}}\,\mathsf{Id}^2$$

$$\mathsf{lambda} \;:\; \mathsf{ExprF}\,\mathsf{IExprF} \longrightarrow \mathsf{IExprF}\,\mathsf{ExprF}$$

Distributive laws can be combined in various ways, e.g., by $\oplus$ and $\otimes$ defined as follows:

Definition 4. *Let* $\lambda_i \;:\; F_iB \longrightarrow BF_i,\ i \in \{1,2\}$ *be distributive laws.*
Then we define a distributive law:

$$\lambda_1 \oplus \lambda_2 \;:\; (F_1 + F_2)B \longrightarrow B(F_1 + F_2)$$
$$\lambda_1 \oplus \lambda_2 \equiv (B\iota_1 \triangledown B\iota_2) \circ (\lambda_1 + \lambda_2)$$

Here, $f \triangledown g$ *is the* cotuple *of* f *and* g *with injections* ι_1 *and* ι_2*, that is the unique arrow such that* $(f \triangledown g) \circ \iota_1 \;=\; f$ *and* $(f \triangledown g) \circ \iota_2 \;=\; g$*.*

[4] Note that we use just juxtaposition for functor composition. Confusion with application is not an issue because application can be always considered as composition with a constant functor. Also note that $F^2 \cong F \times F$ in a bicartesian closed category.

Definition 5. *Let* $\lambda_i : FB_i \longrightarrow B_iF,\ i \in \{1,2\}$ *be distributive laws. Then we define a distributive law:*

$$\lambda_1 \otimes \lambda_2 : F(B_1 \times B_2) \longrightarrow (B_1 \times B_2)F$$
$$\lambda_1 \otimes \lambda_2 \equiv (\lambda_1 \times \lambda_2) \circ (F\pi_1 \vartriangle F\pi_2)$$

Here, $f \vartriangle g$ *is the* tuple *of* f *and* g *with projections* π_1 *and* π_2*, that is the unique arrow such that* $\pi_1 \circ (f \vartriangle g) = f$ *and* $\pi_2 \circ (f \vartriangle g) = g$*.*

We assume the usual convention that $\otimes$ binds stronger than $\oplus$.

Example 5. As algebras and coalgebras are distributive laws, $\oplus$ and $\otimes$ readily specialize to combinators on algebras and coalgebras, as in bothAlg or eitherCoalg. A nontrivial example of a combination of distributive laws is lambda:

$$\mathsf{lambda} = \mathsf{numEval} \otimes \mathsf{numModn} \oplus \mathsf{addEval} \otimes \mathsf{addModn} \tag{5}$$

The following lemma states a basic algebraic property of $\oplus$ and $\otimes$.

Lemma 1. *Let* $\lambda_{i,j} : F_{i,j}B_{i,j} \longrightarrow B_{i,j}F_{i,j},\ i,j \in \{1,2\}$ *be distributive laws. Then:*

$$(\lambda_{1,1} \otimes \lambda_{2,1}) \oplus (\lambda_{1,2} \otimes \lambda_{2,2}) = (\lambda_{1,1} \oplus \lambda_{1,2}) \otimes (\lambda_{2,1} \oplus \lambda_{2,2})$$

Proof. By elementary properties of tuples and cotuples, in particular, by the following law (called the "abides law" in [19]):

$$(f \vartriangle g) \triangledown (h \vartriangle i) = (f \triangledown h) \vartriangle (g \triangledown i)$$

□

Example 6. By the above lemma (compare with (5)):

$$\mathsf{lambda} = (\mathsf{numEval} \oplus \mathsf{addEval}) \otimes (\mathsf{numModn} \oplus \mathsf{addModn}) \tag{6}$$

Examples 5 and 6 illustrate the duality between the functional and OO approaches to *program decomposition*. In functional programming, different cases of the same function, each for a different component of the top-level disjoint union of an algebraic data type, are cotupled by case distinction (c.f. occurrences of $\oplus$ in (6)). In contrast, in OO programming, functions on the same data are tupled into object types (c.f. occurrences of $\otimes$ in (5)).

3.4 The simple expression lemma

We have shown that we may extract a natural transformation from the algebra of a functional fold that can be reused in the coalgebra of an unfold for object construction. It remains to be shown that the functional fold and the OO unfold are indeed semantically equivalent in such a case.

Given a distributive law $\lambda : FB \longrightarrow BF$, one can define an arrow $\mu F \longrightarrow \nu B$ by the following derivation:

$$\lambda_{\nu B} \circ F\mathsf{out}_B : F\nu B \longrightarrow BF\nu B \tag{7}$$

$$[\!(\, \lambda_{\nu B} \circ F\mathsf{out}_B \,)\!]_B : F\nu B \longrightarrow \nu B \tag{8}$$

$$(\!|\, [\!(\, \lambda_{\nu B} \circ F\mathsf{out}_B \,)\!]_B \,|\!)_F : \mu F \longrightarrow \nu B \tag{9}$$

An example of (7) is eitherCoalg in § 2.5.
Dually, the following also defines an arrow $\mu F \longrightarrow \nu B$:

$$B\mathsf{in}_F \circ \lambda_{\mu F} : FB\mu F \longrightarrow B\mu F \tag{10}$$

$$(\!|\, B\mathsf{in}_F \circ \lambda_{\mu F} \,|\!)_F : \mu F \longrightarrow B\mu F \tag{11}$$

$$[\!(\, (\!|\, B\mathsf{in}_F \circ \lambda_{\mu F} \,|\!)_F \,)\!]_B : \mu F \longrightarrow \nu B \tag{12}$$

An example is bothAlg in § 2.5.

The following theorem shows that (12) is equal to (9) and thus, as discussed in § 2.5, establishes a formal correspondence between anamorphically phrased OO programs and catamorphically phrased functional programs.

Theorem 1 ("Simple expression lemma"). *Let* out_B *be the terminal* B*-coalgebra and* in_F *be the initial* F*-algebra. Let* $\lambda : FB \longrightarrow BF$*. Then*

$$(\!|\, [\!(\, \lambda_{\nu B} \circ F\mathsf{out}_B \,)\!]_B \,|\!)_F \quad = \quad [\!(\, (\!|\, B\mathsf{in}_F \circ \lambda_{\mu F} \,|\!)_F \,)\!]_B$$

Proof. We show that the right-hand side, $[\!(\, (\!|\, B\mathsf{in}_F \circ \lambda_{\mu F} \,|\!)_F \,)\!]_B$, satisfies the universal property of the left-hand side; c.f. (2):

$$[\!(\, \lambda_{\nu B} \circ F\mathsf{out}_B \,)\!]_B \circ F[\!(\, (\!|\, B\mathsf{in}_F \circ \lambda_{\mu F} \,|\!)_F \,)\!]_B \quad = \quad [\!(\, (\!|\, B\mathsf{in}_F \circ \lambda_{\mu F} \,|\!)_F \,)\!]_B \circ \mathsf{in}_F \tag{13}$$

The calculation is straightforward by a two-fold application of the following rule, called "AnaFusion" in [19]:

$$[\!(\, \varphi \,)\!]_B \circ f \;=\; [\!(\, \psi \,)\!]_B \quad \Leftarrow \quad \varphi \circ f \;=\; Bf \circ \psi \tag{14}$$

The premise of the rule is precisely the statement that f is a B-coalgebra morphism to φ from ψ. The proof is immediate by compositionality of coalgebra morphisms and uniqueness of the universal arrow. Using this rule we proceed as follows:

$$\begin{array}{ll}
 & [\!(\, \lambda \circ F\mathsf{out}_B \,)\!]_B \circ F[\!(\, (\!|\, B\mathsf{in}_F \circ \lambda \,|\!)_F \,)\!]_B \\
= & \quad \{ \text{ By (14) and the following:} \\
 & \quad \lambda \circ F\mathsf{out}_B \circ F[\!(\, (\!|\, B\mathsf{in}_F \circ \lambda \,|\!)_F \,)\!]_B \\
 & = \quad \{ \text{ functor composition } \} \\
 & \quad \lambda \circ F(\mathsf{out}_B \circ [\!(\, (\!|\, B\mathsf{in}_F \circ \lambda \,|\!)_F \,)\!]_B) \\
 & = \quad \{ [\!(\ldots)\!]_B \text{ is a coalgebra morphism } \} \\
 & \quad \lambda \circ F(B[\!(\, (\!|\, B\mathsf{in}_F \circ \lambda \,|\!)_F \,)\!]_B \circ (\!|\, B\mathsf{in}_F \circ \lambda \,|\!)_F) \\
 & = \quad \{ \lambda \text{ is natural } \} \\
 & \quad (BF[\!(\, (\!|\, B\mathsf{in}_F \circ \lambda \,|\!)_F \,)\!]_B) \circ \lambda \circ F(\!|\, B\mathsf{in}_F \circ \lambda \,|\!)_F \quad \}
\end{array}$$

$$
\begin{array}{ll}
 & [\!(\lambda \circ F(\!| B\mathsf{in}_F \circ \lambda |\!)_F)\!]_B \\
= & \quad \{ \text{ By (14) and the following fact:} \\
 & \qquad (\!| B\mathsf{in}_F \circ \lambda |\!)_F \circ \mathsf{in}_F \\
 & \qquad = \quad \{ \; (\!| \ldots |\!)_F \text{ is a } F\text{-algebra morphism } \} \\
 & \qquad B\mathsf{in}_F \circ \lambda \circ F(\!| B\mathsf{in}_F \circ \lambda |\!)_F \quad \} \\
 & [\!((\!| B\mathsf{in}_F \circ \lambda |\!)_F)\!]_B \circ \mathsf{in}_F
\end{array}
$$

□

4 Classes of dualizable folds

The present section illustrates important classes of folds that are covered by the formal development of this paper (including the elaboration to be expected from § 5). For each class of folds, we introduce a variation on the plain fold operation so that the characteristics of the class are better captured. The first argument of a varied fold operation is not a fold algebra formally; it is rather used for composing a proper fold algebra, which is to be passed to the plain fold operation.

4.1 Void folds

Consider again the Java rendering of the modn function:[5]

```
public abstract void Expr.modn(int v); //Modify literals modulo v
public void Num.modn(int v) { this.value = this.value % v; }
public void Add.modn(int v) { left.modn(v); right.modn(v); }
```

That is, the modn method needs to mutate the value fields of all Num objects, but no other changes are needed. Hence, the imperative OO style suggests to defer to a method without a proper result type, i.e., a void method. In the reading of functional objects, a void method has a result type that is equal to the type parameter of the interface functor. There is a restricted fold operation that captures the idea of voidity in a functional setup:

```
type VExprArg x =
       (Int → x → Int,
        (x → Expr, x → Expr) → x → (Expr, Expr))
vFoldExpr :: VExprArg x →Expr →x → Expr
vFoldExpr a = foldExpr (((.) num . fst a) \/ ((.) add . snd a))
```

The void fold operation takes a product — one type-preserving function for each data variant. The type parameter x enables void folds with extra arguments. For instance, the modn function can be phrased as a void fold with an argument of type Int:

```
modn :: Expr → Int → Expr
modn = vFoldExpr (numModn, addModn)
```

[5] We use a concise OO notation such that the hosting class of an instance method is simply shown as a qualifier of the method name when giving its signature and implementation.

Rewriting a general fold to a void fold requires nothing more than factoring the general fold algebra so that it follows the one that is composed in the vFoldExpr function above. That is, for each constructor, its case preserves the constructor. A void fold must also be sufficiently natural in order to be dualizable; c.f. the next subsection.

4.2 Natural folds

Consider again the type of the fold algebra for the modn function as introduced in § 2.4:

```
   modnAlg :: ExprAlg (Int → Expr)
≡ modnAlg :: ExprF (Int → Expr) → (Int → Expr)
```

The type admits observation of the precise structure of intermediate results; c.f. the occurrences of Expr. This capability would correspond to unlimited introspection in OO programming (including "instance of" checks and casts). The formal development requires that the algebra must be amenable to factoring as follows:

```
modnAlg = fmapIExprF InExpr . lambda
 where
  lambda :: ExprF (Int → x) → Int → ExprF x
  lambda = ...
```

That is, the algebra is only allowed to observe one layer of functorial structure. In fact, it is easy to see that the actual definition of modnAlg suffices with this restriction. We may want to express that a fold is readily in a "natural form". To this end, we may use a varied fold operation whose argument type is accordingly parametric:[6]

```
type NExprArg x = forall y. ExprF (x → y) → x → ExprF y
nExpr :: NExprArg x →Expr →x → Expr
nExpr a = foldExpr ((.) InExpr . a)
```

It is clear that the type parameter x is used at the type Expr, but universal quantification rules out any exploitation of this fact, thereby enabling the factoring that is required by the formal development. For completeness' sake, we also provide a voidity-enforcing variation; it removes the liberty of replacing the outermost constructor:

```
type VnExprArg x = forall y. (Int→x→Int,(x→y,x→y)→x→(y,y))
vnExpr :: VnExprArg x →Expr →x → Expr
vnExpr a = foldExpr ((.) InExpr . (((.) Left . fst a) \/ ((.) Right . snd a)))
```

The modn function is a void, natural fold:

```
modn :: Expr → Int → Expr
modn = vnExpr (numModn, addModn)
```

The distributive law of the formal development, i.e., $\lambda : FB \longrightarrow BF$, is more general than the kind of natural F-folds that we illustrated above. That is, λ is not limited to type-preserving F-folds, but it uses the extra functor B to define the result type of the F-fold in terms of F. This extra functor is sufficient to cover "constant folds" (such as eval), "paramorphic folds" [18] (i.e., folds that also observe the unprocessed, immediate components) and "tupled folds" (i.e., folds that were composed from separated folds by means of tupling). The source distribution of the paper illustrates all these aspects.

[6] We use a popular Haskell 98 extension for rank-2 polymorphism; c.f. `forall`.

4.3 Free monadic folds

The type of the natural folds considered so far implies that *intermediate results are to be combined in exactly one layer of functorial structure*, c.f. the use of ExprF in the result-type position of NExprArg. The formal development will waive this restriction in § 5. Let us motivate the corresponding generalization.

The generalized scheme of composing intermediate results is to arbitrarily nest constructor applications, including the base case of returning one recursive result, as is. Consider the following function that returns the leftmost expression; its Add case does not replace the outermost constructor; instead, the outermost constructor is dropped:

```
leftmost :: Expr → Expr
leftmost (Num i) = Num i
leftmost (Add l r) = leftmost l
```

The function cannot be phrased as a natural fold. We can use a plain fold, though:

```
leftmost = foldExpr (num \/ fst)
```

The following OO counterpart is suggestive:

```
public abstract Expr Expr.leftmost();
public Expr Num.leftmost() { return this; }
public Expr Add.leftmost() { return left.leftmost(); }
```

Consider the following function for "exponential cloning"; it combines intermediate results in a *nest* of constructor applications (while the leftmost function dropped off a constructor):

```
explode :: Expr → Expr
explode x@(Num i) = Add x x
explode (Add l r) = clone (Add (explode l) (explode r))
  where clone x = Add x x
```

Again, the function cannot be phrased as a natural fold. We can use a plain fold:

```
explode :: Expr → Expr
explode = fFoldExpr ((clone . freeNum) \/ (clone . freeAdd . (var <*> var)))
  where clone = freeAdd . (id /\ id)
```

The following OO counterpart uses the "functional" constructors for the object types.

```
public abstract Expr Expr.explode();
public Expr Num.explode() { return new Add(new Num(value),new Num(value)); }
public Expr Add.explode() {
        return new Add(
                new Add(left.explode(), right.explode()),
                new Add(left.explode(), right.explode()));
}
```

We need a generalized form of natural F-folds where the result of each case may be either of type x, or of type $F\,x$, or of any type $F^n\,x$ for $n \geq 2$. The corresponding union of types is modeled by the free type of F, i.e., the type of free terms (variable terms) over F. This type is also known as the free monad. For instance, the free type of expressions is defined as follows:

```
newtype FreeExpr x = InFreeExpr { outFreeExpr :: Either x (ExprF (FreeExpr x)) }
```

We also assume the following convenience injections:

```
var  = InFreeExpr . Left
term = InFreeExpr . Right
freeNum = term . Left
freeAdd = term . Right
```

Natural folds are generalized as follows:

```
type FExprArg x = forall x. ExprF x → FreeExpr x
fFoldExpr :: FExprArg x →Expr → Expr
fFoldExpr a = foldExpr (collapse . a)
  where
    collapse :: FreeExpr Expr →Expr
    collapse = (id \/ (leaf \/ (fork . (collapse <*> collapse)))) . outFreeExpr
```

(For simplicity, we only consider folds without extra arguments here.) That is, we compose together a fold algebra that first constructs free-type terms from intermediate results, and then collapses the free-type layers by conversion to the recursive closure of the functor at hand. Term construction over the free type is to be dualized to object construction. The two motivating examples can be expressed as "free" natural folds:

```
leftmost  = fFoldExpr (freeNum \/ (var . fst ))
explode = fFoldExpr ((clone . freeNum) \/ (clone . freeAdd . (var <*> var)))
  where clone = freeAdd . (id /\ id)
```

4.4 Cofree comonadic folds

The type of the natural folds considered so far implies that *the algebra has access to the results of applying the recursive function to immediate components exactly once*. The formal development will waive this restriction in § 5. Let us motivate the corresponding generalization.

In general, we may want to apply the recursive function any number of times to each immediate component. We may think of such expressiveness as an iteration capability. Here is a very simple example of a function that increments twice on the left, and once on the right:

```
leftist  :: Expr → Expr
leftist  (Num i) = Num (i+1)
leftist  (Add l r) = Add (leftist (leftist l)) (leftist r)
```

Such iteration is also quite reasonable in OO programming:

```
public abstract void Expr.leftist ();
public void Num.leftist () { value++; }
public void Add.leftist () { left.leftist (); left.leftist (); right.leftist (); }
```

The simple example only involves a single recursive function, and hence, arbitrary repetitions of that function can be modeled as a *stream* (i.e., a coinductive list). A designated, streaming-enabled fold operation, sFoldExpr, deploys a stream type as the result

type. The argument of such a fold operation can select among different numbers of repetitions in the following style:

```
leftist :: Expr → Expr
leftist = sFoldExpr ((+1) <|> ((head . tail . tail) <*> (head . tail)))
```

Here, head maps to "0 applications", head . tail maps to "1 application", head . tail . tail maps to "2 applications". The streaming-enabled fold operation composes together a fold algebra that produces a stream of results at each level of folding. To this end, we need the coinductive unfold operation for streams:

```
type StreamCoalg x y = y →(x,y)
unfoldStream :: StreamCoalg x y →y → [x]
unfoldStream c = uncurry (:) . (id <*> unfoldStream c) . c
```

The streaming-enabled fold operation is defined as follows:

```
type SExprArg = forall x. ExprF [x] → ExprF x
sFoldExpr :: SExprArg →Expr →Expr
sFoldExpr a = head . tail . foldExpr a'
    where
      a' :: ExprF [Expr] → [Expr]
      a' = unfoldStream (hd /\ tl)
      hd = InExpr . (id <|> (head <*> head))
      tl = a . (id <|> (iterate tail <*> iterate tail))
```

The argument type of the operation, SExprArg, can be interpreted as follows: given a stream of repetitions for each immediate component, construct a term with a selected number of repetitions for each immediate component position. In fact, the selection of the favored number of repetitions is done by cutting of only the disfavored prefix of the stream (as opposed to actual selection, which would also cut off the postfix). Thereby, iteration is enabled; each step of iteration further progresses on the given stream.

If we look closely, we see that the argument a :: SExprArg is not provided with a *flat* stream of repetitions but rather *a stream of streams* of remaining repetitions; c.f. the use of the standard function, iterate, for coinductive iteration. The type SExprArg protects the nesting status of the stream by universal quantification, thereby avoiding that the stream of remaining repetitions is manipulated in an undue manner such as by reshuffling. For comparison, an unprotective type would be the following:

```
forall x. ExprF [[x]] → ExprF [x]
```

When multiple (mutually recursive) functions are considered, then we must turn from a stream of repetitions to infinite trees of repeated applications; each branch corresponds to the choice of a particular mutation or observation. This tree structure would be modeled by a functor, reminiscent of an interface functor, and the stream type is generalized to the cofree comonad over a functor.

5 The categorical model continued

We will now extend the theory of § 3 in order to cater for the examples of § 4. In particular our notion of a dualizable program, which used to be a simple natural transfor-

mation of type $FB \longrightarrow BF$, will be extended to more elaborate distributive laws between a monad and a comonad [2,25]. This extra structure provides the power needed to cover functional folds that iterate term construction or recursive function application. We show that all concepts from § 3 lift appropriately to (co)monads. We also provide means to construct the generalized distributive laws from more manageable natural transformations, which are the key to programming with the theory and justify the examples of § 4. We eventually generalize the final theorem of § 3 about the semantical correspondence between functions and objects. The used categorical tools are relatively straightforward and well known; [1] is an excellent reference for this section. We only claim originality in their adoption to the semantical correspondence problem of the expression lemma.

5.1 ((Co)free) (co)monads

We begin with a reminder of the established definitions of (co)monads.

Definition 6. *Let* $\mathbf{C}$ *be a category. A* ***monad*** *on* $\mathbf{C}$ *is a triple* $\langle T, \mu, \eta \rangle$*, where* T *is an endofunctor on* $\mathbf{C}$*,* $\eta : \mathsf{Id}_{\mathbf{C}} \longrightarrow T$ *and* $\mu : T^2 \longrightarrow T$ *are natural transformations satisfying the following identities:*

$$\mu \circ \mu_T = \mu \circ T\mu \tag{15}$$
$$\mu \circ \eta_T = \mathsf{id} = \mu \circ T\eta \tag{16}$$

A ***comonad*** *is defined dually as a triple* $\langle D, \delta, \epsilon \rangle$ *where* $\delta : D \longrightarrow D^2$ *and* $\epsilon : D \longrightarrow \mathsf{Id}_{\mathbf{C}}$ *satisfy the duals of (15) and (16):*

$$\delta_D \circ \delta = D\delta \circ \delta \tag{17}$$
$$\epsilon_D \circ \delta = \mathsf{id} = D\epsilon \circ \delta \tag{18}$$

To match § 4.3 and § 4.4, we need (co)free (co)monads.

Definition 7. *Let* F *be an endofunctor on* $\mathbf{C}$*. Let* $F^\dagger_X$ *be the functor* $F^\dagger_X = X + F\mathsf{Id}$*. The* ***free monad*** *of the functor* F *is a functor* T_F *that is defined as follows:*

$$T_F\, X = \mu F^\dagger_X$$
$$T_F\, f = (\!|\, \mathsf{in}_{F^\dagger_Y} \circ (f + \mathsf{id}) \,|\!)_{F^\dagger_X} \quad , \text{ for } f : X \longrightarrow Y$$

We make the following definitions:

$$\eta_X = \mathsf{in}_{F^\dagger_X} \circ \iota_1 : X \longrightarrow T_F\, X$$
$$\tau_X = \mathsf{in}_{F^\dagger_X} \circ \iota_2 : FT_FX \longrightarrow T_FX$$
$$\mu_X = (\!|\, \mathsf{id}_{T_FX} \triangledown \tau_X \,|\!)_{F^\dagger_{T_X}}$$

Now, η *and* μ *are natural transformations,* $\langle T_F, \mu, \eta \rangle$ *is a monad.*

Example 7. We can think of the type $T_F X$ as of the type of non-ground terms generated by signature F with variables from X; c.f. the Haskell data type FreeExpr in § 4.3. Then, η makes a variable into a term (c.f. var); τ constructs a term from a constructor in F applied to terms (c.f. term), μ is substitution (c.f.collapse, which is actually μ_0; the type collapse : ExprF Expr → Expr reflects the fact that $T_F 0 \cong \mu F$).

Cofree comonads are defined dually.

Definition 8. *Let B be an endofunctor on $\mathbf{C}$, let $B^{\ddagger}_X$ be the functor $B^{\ddagger}_X = X \times B\mathsf{Id}$. The **cofree comonad** of the functor B is a functor D_B that is defined as follows:*

$$\begin{aligned} D_B\, X &= \nu B^{\ddagger}_X \\ D_B\, f &= [\!((f \times \mathsf{id}) \circ \mathsf{out}_{B^{\ddagger}_X})\!]_{B^{\ddagger}_Y} \\ \epsilon_X &= \pi_1 \circ \mathsf{out}_{B^{\ddagger}_X} : D_B\, X \longrightarrow X \\ \xi_X &= \pi_2 \circ \mathsf{out}_{B^{\ddagger}_x} : D_B X \longrightarrow B D_B X \\ \delta_X &= [\!(\mathsf{id}_{D_B X} \vartriangle \xi_X)\!]_{B^{\ddagger}_{D_X}} \end{aligned}$$

Example 8. Let us consider a special case: $D_{\mathsf{Id}}\, X$. We can think of this type as the stream of values of type X; c.f. the coinductive use of the Haskell's list-type constructor in § 4.4. This use corresponds to the following cofree comonad:

$$\mathsf{Stream}\ X \ = \ \nu Y.X \times Y \ = \ D_{\mathsf{Id}}\, X$$

Here ϵ is head, ξ is tail, δ is tails : stream $X \to$ stream (stream X): the function turning a stream into the stream of its tails, i.e. iterate tail. Also note that $D_B 1 \cong \nu B$.

A note on free and non-free monads and comonads In the present paper, we deal exclusively with free constructions (monads and comonads); free constructions have straightforward interpretations as programs. However part of the development, and Theorem 5 in particular, hold in general, and there are interesting examples of non-free constructions arising for instance as quotients with respect to collections of equations.

5.2 (Co)monadic (co)algebras and their morphisms

The notions of folds (and algebras) and unfolds (and coalgebras) lift to monads and comonads. This status will eventually allows us to introduce distributive laws of monads over comonads. We begin at the level of algebras. An algebra of a monad is an algebra of the underlying functor of the monad, which respects the unit and multiplication. Thus:

Definition 9. *Let $\mathbf{C}$ be a category, let $\langle T, \eta, \mu \rangle$ be a monad on $\mathbf{C}$. An T-algebra is an arrow $\alpha : TX \longrightarrow X$ in $\mathbf{C}$ such that the following equations hold:*

$$\mathsf{id}_X \ = \ \alpha \circ \eta_X \tag{19}$$

$$\alpha \circ \mu_X \ = \ \alpha \circ \ T\alpha \tag{20}$$

A T-algebra morphism between T-algebras $\alpha : TA \longrightarrow A$ and $\beta : TB \longrightarrow B$ is an arrow $f : A \longrightarrow B$ in $\mathbf{C}$ such that

$$f \circ \alpha \;=\; \beta \circ Tf$$

The category of T-algebras for a monad[7] T is denoted $\mathbf{C}^T$.

The notion of the *category of D-coalgebras*, $\mathbf{C}_D$, for a *comonad* D is exactly dual. We record the following important folklore fact about the relation of (co)algebras of a functor and (co)algebras of its free (co)monad; it allows us to compare the present (co)monadic theory to the simple theory of § 3.

Theorem 2. *Let F and B be endofunctors on $\mathbf{C}$. Then the category $\mathbf{C}^F$ of F-algebras is isomorphic to the category $\mathbf{C}^{T_F}$ of algebras of the free monad. And dually: the category $\mathbf{C}_B$ of B-coalgebras is isomorphic to $\mathbf{C}_{D_B}$, the category of coalgebras of the cofree comonad D_B.*

Proof. We show the two constructions: from $\mathbf{C}^F$ to $\mathbf{C}^{T_F}$ and back. To this end, let $\varphi : FX \longrightarrow X$ be an F-algebra, then $\varphi^* \;=\; (\!|\, \mathsf{id}_X \triangledown \varphi \,|\!)_{F^\dagger_X}$ is a T_F-algebra. (Read φ with superscript $*$ as "lift φ freely".) In the other direction, given a T_F-algebra α, the arrow $\alpha_* \;=\; \alpha \circ \tau_X \circ F\eta_X$ is an F-algebra. (Read α with subscript $*$ as "unlift α freely".) We omit the check that the two directions are inverse and that φ^* is indeed a T_F-algebra. The dual claim follows by duality: for $\psi : X \longrightarrow BX$, $\psi^\propto \;=\; [\![\, \mathsf{id}_X \;\vartriangle\; \psi \,]\!]_{B^\ddagger_X}$; $\beta_\propto \;=\; B\epsilon_X \circ \xi_X \circ \beta$.

Example 9. Consider the functor ExprF. Then $T_{\mathsf{ExprF}}\, X$ is the type of expressions with (formal) variables from X. For $\mathsf{evalAlg} \;\equiv\; \mathsf{id} \triangledown (\mathsf{curry}\,(+)) \;:\; \mathsf{Int} + \mathsf{Int}^2 \longrightarrow \mathsf{Int}$, $\mathsf{evalAlg}^* \;:\; T_{\mathsf{ExprF}}\, \mathsf{Int} \longrightarrow \mathsf{Int}$ recursively evaluates an expression where the variables are integers. For $\mathsf{eval}' \;:\; T_{\mathsf{ExprF}}\, \mathsf{Int} \longrightarrow \mathsf{Int}$ that evaluates ("free") expressions, one can reproduce the function $\mathsf{evalAlg} \;\equiv\; \mathsf{eval}'_*$ by applying eval' to trivial expressions, provided that eval' is sufficiently uniform (in the sense of equations (19) and (20)).

Example 10. Remember that $D_{\mathsf{Id}} \cong \mathsf{Stream}$. For an Id-coalgebra $k \;:\; X \longrightarrow X$, $k^\propto \;=\; \mathsf{iterate} \;:\; X \longrightarrow \mathsf{Stream}\, X$. And obviously, from a function $\mathsf{itf} \;:\; X \longrightarrow \mathsf{stream}\, X$ producing a stream of iterated results of a function, one can reproduce the original function $\mathsf{itf}_\propto$ by taking the second element of the stream.

The theorem, its proof, and the examples illustrate the key operational intuition about algebras of free monads: any algebra of a free monad works essentially in an iterative fashion. It is equivalent to the iteration of a plain algebra, which processes (deep) terms by induction on their structure. If we consider a T_F-algebra $\alpha \;:\; T_F X \longrightarrow X$ as a function reducing a tree with leaves from X into a single X, α_* is the corresponding one-layer function which prescribes how each layer of the tree is collapsed given an operator and a collection of collapsed subtrees. This intuition is essential for building an intuition about generalized distributive laws, which are to come shortly, and the programs induced by them.

We continue lifting concepts from § 3.

[7] We are overloading the notation here as a monad is also a functor. The convention is that when T is a monad, $\mathbf{C}^T$ is the category of algebras of the *monad*.

Lemma 2. *Let 0 be the initial object in $\mathbf{C}$. Then μ_0 is the initial T-algebra in $\mathbf{C}^T$. Dually, δ_1 is the terminal D-coalgebra in $\mathbf{C}_D$.*

Proof. Omitted.

We can now lift the definitions of folds and unfolds.

Definition 10. *Let α be a T-algebra. Then $(\!|\, \alpha \,|\!)_T$ denotes the unique arrow in $\mathbf{C}$ from the initial T-algebra to α. Dually, for a D-coalgebra β and $[\!(\, \beta \,)\!]_D$.*

$$h = (\!|\, \alpha \,|\!)_{T_F} \Leftrightarrow h \circ \mu_0 = \alpha \circ Fh, \quad \textit{and } \alpha \textit{ is a } T\textit{-algebra} \tag{21}$$

$$h = [\!(\, \beta \,)\!]_F \Leftrightarrow \delta_1 \circ h = Fh \circ \beta, \quad \textit{and } \beta \textit{ is a } D\textit{-coalgebra} \tag{22}$$

5.3 Distributive laws of monads over comonads

Distributive laws of monads over comonads, due to J. Beck [2], are liftings of the plain distributive laws of § 3, which had a straightforward computational interpretation, to monads and comonads. Again, they are natural transformations on the functors, but they respect the additional structure of the monad and comonad in question.

The following is standard, here taken from [25].

Definition 11. *Let $\langle T, \eta, \mu \rangle$ be a monad and $\langle D, \varepsilon, \delta \rangle$ be a comonad in a category $\mathcal{C}$. A **distributive law** of T over D is a natural transformation*

$$\Lambda \;:\; TD \longrightarrow DT$$

satisfying the following:

$$\Lambda \circ \eta_D = D\eta \tag{23}$$

$$\Lambda \circ \mu_D = D\mu \circ \Lambda_T \circ T\Lambda \tag{24}$$

$$\varepsilon_T \circ \Lambda = T\varepsilon \tag{25}$$

$$\delta_T \circ \Lambda = D\Lambda \circ \Lambda_D \circ T\delta \tag{26}$$

In the following, we relate "programming" to distributive laws, where we further emphasize the view that programs are represented as natural transformations. First, we show that natural transformations with uses of monads and comonads give rise to distributive laws of monads over comonads; see the following theorem. In this manner, we go beyond the simple natural transformations and distributive laws of § 3, and hence we can dualize more programs.

Theorem 3 (Plotkin and Turi, 1997). *Let F and B be endofunctors. Natural transformations of type*

$$F(\mathsf{Id} \times B) \longrightarrow BT_F \tag{27}$$

or

$$FD_B \longrightarrow B(\mathsf{Id} + F) \tag{28}$$

give rise to distributive laws $\Lambda : T_F D_B \longrightarrow D_B T_F$.

Proof. See [25]. □

The theorem originated in the context of categorical operational semantics [25]; see the related work discussion in § 6. However, the theorem directly applies to our situation of "programming with natural transformations". In the Haskell-based illustrations of § 4, we encountered such natural transformations as arguments of the enhanced fold operations. We did not exhaust the full generality of the typing scheme that is admitted by the theorem, but we did have occurrences of a free monad and a cofree comonad. Here we note that the general types of natural transformations in the above theorem admit paramorphisms [18] (c.f. $F(\mathsf{Id} \times B)$ in (27)) and their duals, *apomorphisms* [27] (c.f. $B(\mathsf{Id} + F)$ in (28)).

Example 11. The type FExprArg in § 4.3 models natural transformations of type $FB \longrightarrow BT_F$ for $F \equiv \mathsf{ExprF}$ and $B \equiv \mathsf{Id}$. Likewise, The type SExprArg in § 4.4 models natural transformations of type $FD_B \longrightarrow BF$, for $F \equiv \mathsf{ExprF}$ and $B \equiv \mathsf{Id}$.

Natural transformations $\lambda : FB \longrightarrow BF$ can be lifted so that Theorem 3 applies:

$$B\tau \circ \lambda_{T_F} \circ FB\eta \circ F\pi_2 : F(\mathsf{Id} \times B) \longrightarrow BT_F \tag{29}$$

$$B\iota_2 \circ BF\epsilon \circ \lambda_{D_B} \circ F\xi : FD_B \longrightarrow B(\mathsf{Id} + F) \tag{30}$$

The following fact is useful:

Lemma 3. *Given a natural transformation $\lambda : FB \longrightarrow BF$, the distributive law constructed by Theorem 3 from (29) is equal to the one constructed from (30).*

Proof. Omitted. See the full report.

The following definition is therefore well-formed.

Definition 12. *For a natural transformation $\lambda : FB \longrightarrow BF$, we denote by $\bar{\lambda}$ the distributive law $T_F D_B \longrightarrow D_B T_F$ given by Theorem 3 either from (29) or (30).*

5.4 Conservativeness of free distributive laws

We can lift simple distributive laws of § 3 to distributive laws of monads over comonads. It remains to establish that such a lifting of distributive laws is semantics-preserving. The gory details follow.

In the simple development of § 3, we constructed algebras and coalgebras from distributive laws by simple projections and injections; c.f. equations (10) and (7). These constructions are lifted as follows:

Lemma 4. *For all X in* **C***, the arrow*

$$D\mu_X \circ \Lambda_{TX} : TDTX \longrightarrow DTX \tag{31}$$

is a T-algebra. Dually, the arrow

$$\Lambda_{DX} \circ T\delta_X : TDX \longrightarrow DTDX \tag{32}$$

is a D-coalgebra.

Proof. We must verify that the two T-algebra laws (19) and (20) hold. This can be done by a simple calculation involving just naturality and definitions. □

Now (31) is a T-algebra, and thus by Lemma 2 and Def. 10 it induces an arrow:

$$(\!| D\mu_0 \circ \Lambda_{T0} |\!)_T \ : \ T0 \longrightarrow DT0 \tag{33}$$

Moreover, when T and D are free on F and B respectively, this is equivalent to

$$(\!| D\mu_0 \circ \Lambda_{T0} |\!)_T \ : \ \mu F \longrightarrow D\mu F$$

which is by Theorem 2 isomorphic to a B-coalgebra

$$(\!| D\mu_0 \circ \Lambda_{T0} |\!)_{T_F \propto} \ : \ \mu F \longrightarrow B\mu F \tag{34}$$

Dually for (32):

$$[\!| \Lambda_{D1} \circ T\delta_1 |\!]_D \ : \ TD1 \longrightarrow D1 \tag{35}$$

$$[\!| \Lambda_{D1} \circ T\delta_X |\!]_{D_B *} \ : \ F\nu B \longrightarrow \nu B \tag{36}$$

Compare with equations (11) and (8). This shows that *any* distributive law of a free monad over a cofree comonad also gives rise to an algebra for a catamorphisms and a coalgebra for object construction.

Example 12. The functions sFoldExpr in § 4.4, and fFoldExpr in § 4.3 are examples of (34) where B in (34) is fixed to be the identity functor.

The following theorem establishes the essential property that the (co)monadic development of the present section entails the development of § 3.

Theorem 4. *Let F and B be endofunctors on a category* **C**. *Let $\lambda : FB \longrightarrow BF$ be a natural transformation. Then the following holds.*

$$[\!| \lambda_{\nu B} \circ F\mathsf{out}_B |\!]_B = [\!| \bar{\lambda}_{D_B 1} \circ T_F \delta_X |\!]_{D_B *} \ : \ F\nu B \longrightarrow \nu B \tag{37}$$

$$(\!| B\mathsf{in}_F \circ \lambda_{\mu F} |\!)_F = (\!| D_B \mu_0 \circ \bar{\lambda}_{T_F 0} |\!)_{T_F \propto} \ : \ \mu F \longrightarrow B\mu F \tag{38}$$

Proof. Omitted. See the full report.

5.5 The generalized expression lemma

It remains to lift Theorem 1 (the "simple expression lemma"). As a preparation, we need an analog of the fusion rule.

Lemma 5. *For D-coalgebras α and β:*

$$[\!| \alpha |\!]_D \circ f \ = \ [\!| \beta |\!]_D \quad \Leftarrow \quad \alpha \circ f = Df \circ \beta \tag{39}$$

Proof. Immediate by uniqueness of the terminal morphism, as before. □

Theorem 5 ("Generalized expression lemma"). *Let $\langle T, \eta, \mu\rangle$ be a monad and $\langle D, \eta, \delta\rangle$ be a comonad. Let $\Lambda : TD \longrightarrow DT$ be a distributive law of the monad T over D. Then the following holds:*

$$(\!| [\![\Lambda_{D1} \circ T\delta_1]\!]_D |\!)_T \quad = \quad [\![(\!| D\mu_0 \circ \Lambda_{T0} |\!)_T]\!]_D$$

Proof. The proof has exactly the same structure as that of Theorem 1 except that we have to check at all relevant places that the algebras and coalgebras in question satisfy the additional properties (19) and (20) or their duals, subject to straightforward applications of the monad laws, properties (23) - (26) of distributive laws, and by naturality. We give an outline of the proof of the theorem while omitting the routine checks.

$$\begin{array}{ll} & [\![\Lambda_D \circ T\delta]\!]_D \circ T[\![(\!| D\mu \circ \Lambda_T |\!)_T]\!]_D \\ = & \quad \{ \text{ By (39) } \} \\ & [\![\Lambda_D \circ T(\!| D\mu \circ \Lambda_T |\!)_T]\!]_D \\ = & \quad \{ \text{ By (39) } \} \\ & [\![(\!| D\mu \circ \Lambda_T |\!)_T]\!]_D \circ \mu \end{array}$$

The conclusion follows by (21). □

6 Related work

Functional OO programming

We are not aware of any similar treatment of the correspondence between functional and OO programming. Initially, one would expect some previous work on functional OO programming to be relevant here, such as Moby [8] (an ML-like language with a class mechanism), C# 3.0/VB 9.0/LINQ [3] (the latest .NET languages that incorporate higher-order list-processing functions and more type inference), F# (an ML/OCaml-inspired language that is married with .NET objects), Scala [20] (a Java-derived language with support for functional programming), ML-ART or OCaml [23] (ML with OO-targeting type extensions) — just to mention a few. However, all such work has not revealed the expression lemma. *When functional OO efforts start from a functional language*, then the focus is normally on type-system extensions for subtyping, self, and inheritance, while OO programs are essentially encoded as functional programs, without though relating the encoding results to any "native" functional counterparts. *Dually, when functional OO efforts start from an OO language*, then the focus is normally on translations that eliminate functional idioms, without though relating the translation results to any "native" OO counterpart. (For instance, Scala essentially models a function as a special kind of object.) Our approach specifically leverages the correspondence between functional folds and OO designs based on an idealized composite pattern.

The expression problem

Previous work on the expression problem [28] has at best assumed the expression lemma implicitly. The lemma may have been missing because the expression problem classically assumes only very little structure: essentially, there are supposed to be

multiple data variants as well as multiple operations on these variants. In contrast, the proposed expression lemma requires more structure, i.e., it requires functional folds or OO designs based on an idealized composite pattern, respectively.

Programming vs. semantics

We have demonstrated how distributive laws of a functor over a functor (both possibly with additional structure) arise naturally from programming practice with disciplined folds and an idealized composite pattern. By abstraction, we have ultimately arrived at the same notion of adequacy that Turi and Plotkin originally coined for denotational and operational semantics [24,25]. There, our functional programming side of the picture corresponds to *denotational semantics* and the OO programming side corresponds to *operational semantics*. Our functional/OO programming correspondence corresponds to *adequacy of denotational and operational semantics*. We have provided a simple, alternative, calculational proof geared towards functional programming intuitions. Any further correspondence, for instance of their *operational rules*, is not straightforward. We hypothesize that an elaborated expression lemma may eventually incorporate additional structure that has no direct correspondence in Turi and Plotkin's sense.

More on distributive laws

Distributive laws of a functor over a functor (both possibly with additional structure) [2] have recently enjoyed renewed interest. We mention a few of the more relevant contributions. In the context of bialgebraic semantics, Fiore, Plotkin and Turi have worked on languages with binders in a presheaf category [7], and Bartek Klin has worked on recursive constructs [16]. Both theoretical contributions may inspire a model of object structures with cycles and sharing in our interpretation. Modular constructions on distributive laws, including those we leveraged towards the end of § 3.3 have been investigated by Bart Jacobs [13]. More advanced modular constructions may be helpful in the further exploration of the modularity of dualizable programs. Alberto Pardo, Tarmo Uustalu, Varmo Vene, and collaborators have been using distributive laws for recursion and corecursion schemes. For instance, in [26], a generalized coinduction scheme is delivered where a distributive law specifies the pattern of mutual recursion between several functions defined by coinduction. This work seems to be related to coalgebraic OO programming where methods in an interface are (possibly) mutually recursive.

7 Concluding remarks

We have revealed the expression lemma — a correspondence between OO and functional programs, subject to the assumption that both kinds of programs share a certain decomposition based on structural recursion. The decomposition requirement for functional programs is equivalent to a class of natural folds. The decomposition requirement for OO programs is equivalent to the concept of object structures with part-whole relationships and methods that are in alignment with an idealized composite design pattern.

The formal development for comparing functional and OO programs relies on a coalgebraic model of functional objects.

While our development already covers some non-trivial idioms in "dualizable" programming, e.g., iteration of term construction and recursive function application, it still leaves many open questions — in particular, if we wanted to leverage the duality for real-world programs. Hence, one challenge is to generalize the expression lemma and the associated constructions of distributive laws so that the class of dualizable programs is extended. For instance, histomorphisms [26,15] are not just useful in devising efficient encodings for non-linearly recursive problems, they also generalize access to intermediate results for non-immediate components — very much in the sense of "dotting" into objects and invoking methods on non-immediate components. Another challenge is to admit data structures with sharing and cycles. The use of sharing and cycles is common in OO programming — even without the additional complication of mutable objects. Yet another challenge is to complete the current understanding of the functional/OO correspondence into effective bidirectional refactorings. For instance, in the objects-to-functions direction, such a refactoring would involve non-trivial preconditions on the shape of the OO code, e.g., preconditions to establish absence of sharing, cycles, and mutations, where we may be able to leverage related OO type-system extensions, e.g., for immutability and ownership [6,4].

Acknowledgments Ondrej Rypacek would like to thank the CALCO-jnr 2007 referees for feedback that could be leveraged for the present paper. This research has been funded by EPSRC grant EP/D502632/1. The authors are grateful for the constructive and detailed reviews by the MPC 2008 program committee.

References

1. S. Awodey. *Category Theory*. Clarendon Press, 2006.
2. J. Beck. Distributive laws. *Lecture Notes in Mathematics*, 80:119–140, 1969.
3. G. M. Bierman, E. Meijer, and M. Torgersen. Lost in translation: formalizing proposed extensions to C#. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 479–498. ACM Press, 2007.
4. A. Birka and M. D. Ernst. A practical type system and language for reference immutability. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 35–49. ACM Press, 2004.
5. W.-N. Chin. Towards an automated tupling strategy. In *PEPM '93: Proceedings of the 1993 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 119–132. ACM Press, 1993.
6. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 48–64. ACM Press, 1998.
7. M. Fiore, G. Plotkin, and D. Turi. Abstract Syntax and Variable Binding. In *LICS '99: Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, pages 193–202. IEEE Press, 1999.
8. K. Fisher and J. Reppy. Object-oriented aspects of Moby. Technical report, University of Chicago Computer Science Department Technical Report (TR-2003-10), July 2003.

9. M. M. Fokkinga. Tupling and Mutumorphisms. Appeared in: The Squigollist, Vol 1, Nr 4, 1990, pages 81–82, June 1990.
10. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
11. Z. Hu, H. Iwasaki, and M. Takeichi. Deriving structural hylomorphisms from recursive definitions. In *ICFP '96: Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 73–82. ACM Press, 1996.
12. Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple data traversals. In *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 164–175. ACM Press, 1997.
13. B. Jacobs. Distributive laws for the coinductive solution of recursive equations. *Information and Computation*, 204(4):561–587, 2006.
14. B. P. F. Jacobs. Objects and classes, coalgebraically. In B. Freitag, C. B. Jones, C. Lengauer, and H. J. Schek, editors, *Object-Orientation with Parallelism and Persistence*, pages 83–103. Kluwer Academic Publishers, 1996.
15. J. Kabanov and V. Vene. Recursion Schemes for Dynamic Programming. In T. Uustalu, editor, *Mathematics of Program Construction, 8th International Conference, MPC 2006, Kuressaare, Estonia, July 3-5, 2006, Proceedings*, volume 4014 of *LNCS*, pages 235–252. Springer, 2006.
16. B. Klin. Adding recursive constructs to bialgebraic semantics. *Journal of Logic and Algebraic Programming*, 60-61:259–286, 2004.
17. J. Launchbury and T. Sheard. Warm fusion: deriving build-catas from recursive definitions. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 314–323. ACM Press, 1995.
18. L. G. L. T. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.
19. E. Meijer, M. M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*, volume 523 of *LNCS*, pages 124–144. Springer, 1991.
20. M. Odersky. The Scala Language Specification, Version 2.6, DRAFT, 19 Dec. 2007. Programming Methods Laboratory, EPFL, Switzerland.
21. B. C. Pierce and D. N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, 1994.
22. H. Reichel. An approach to object semantics based on terminal co-algebras. *Mathematical Structures in Computer Science*, 5(2):129–152, 1995.
23. D. Rémy. Programming Objects with ML-ART: An extension to ML with Abstract and Record Types. In M. Hagiya and J. Mitchell, editors, *International Symposium on Theoretical Aspects of Computer Software*, number 789 in LNCS, pages 321–346. Springer, 1994.
24. D. Turi. *Functorial Operational Semantics and its Denotational Dual*. PhD thesis, Free University, Amsterdam, June 1996.
25. D. Turi and G. D. Plotkin. Towards a mathematical operational semantics. In *Proceedings 12th Annual IEEE Symposium on Logic in Computer Science, LICS'97, Warsaw, Poland, 29 June – 2 July 1997*, pages 280–291. IEEE Press, 1997.
26. T. Uustalu, V. Vene, and A. Pardo. Recursion schemes from comonads. *Nordic Journal of Computing*, 8(3):366–390, 2001.
27. V. Vene and T. Uustalu. Functional programming with apomorphisms (corecursion). *Proceedings of the Estonian Academy of Sciences: Physics, Mathematics*, 47(3):147–161, 1998.
28. P. Wadler. The expression problem. Message to java-genericity electronic mailing list, November 1998. Available online at http://www.daimi.au.dk/~madst/tool/papers/expression.txt.