A Notation for Comonads

Dominic Orchard and Alan Mycroft

Computer Laboratory, University of Cambridge
{firstname}.{lastname}@cl.cam.ac.uk

Author's copy. To appear in LNCS volume of the post-proceedings of IFL'12.

Abstract. The category-theoretic concept of a *monad* occurs widely as a design pattern for functional programming with effects. The utility and ubiquity of monads is such that some languages provide syntactic sugar for this pattern, further encouraging its use. We argue that *comonads*, the dual of monads, similarly provide a useful design pattern, capturing notions of context dependence. However, comonads remain relatively under-used compared to monads—due to a lack of knowledge of the design pattern along with the lack of accompanying simplifying syntax. We propose a lightweight syntax for comonads in Haskell, analogous to the do-notation for monads, and provide examples of its use. Via our notation, we also provide a tutorial on programming with comonads.

Many algebraic approaches to programming apply concepts from category theory as design patterns for abstracting and structuring programs. For example, the category-theoretic notion of a *monad* is widely used to structure programs with *side effects*, encapsulating effects within a parametric data type [1, 2]. A *monadic* data type M has accompanying operations which provide composition of functions with *structured output* of type $a \to M\,b$. Side effects can be seen as *impure output* behaviour, encoded by the data type M.

Monads are so effective as an abstraction technique that some languages provide a lightweight syntactic sugar simplifying programming with monads, such as the **do**-notation in Haskell and the *let!* notation in F# [3].

Comonads are the *dual* structure to *monads*, where a comonadic data type C has operations for the composition of functions with *structured input*, of type $C\,a \to b$. Whilst monads capture impure output behaviour (side effects), comonads capture impure *input* behaviour, often described as *context dependence*, encoded by the data type C. There are various examples of programming with comonads in the literature including dataflow programming via streams [4], attribute evaluation [5], array computations [6], and more [7]. However, despite these examples, comonads are less widely used than monads.

There are two reasons for this: one is that they are less well-known, the other, related reason is the lack of language support, which impedes the use of comonads as a design pattern. To remedy this, we propose a syntax which simplifies programming with comonads in Haskell, called the ***codo**-notation*, which also serves to promote the comonad design pattern.

In Haskell, comonads are defined by the following class:[1]

[1] Available via Edward Kmett's `Control.Comonad` package.

```
class Comonad c where
  extract :: c a → a
  extend :: (c a → b) → c a → c b
```

The contextual view of comonads is that values of type $c\ a$ encode context-dependent computations of values of type a, and functions $c\ a \to b$ describe *local operations* within some context. The *extract* operation defines a notion of *current context* and is a trivial local operation returning the value at this context; *extend* defines the range of *all possible contexts*, extending a local operation to a *global* operation by applying it at every context. Thus comonads abstract "boilerplate" code for extending an operation, defined at one context, to all contexts.

For example, arrays can be seen as encoding contextual computations, where a value depends on its position. An array paired with an array index denoting the current context – called the *cursor* – is a comonad. Its *extract* operation accesses the cursor element of the array; *extend* applies a local operation, which computes a value from an array at a particular cursor, to an array at each possible cursor index in its domain (*i.e.*, globally), computing an array of results [6]. Local operations of this form, on arrays, are ubiquitous in image processing, scientific computing, and cellular automata.

The **codo**-notation simplifies programming with comonads. For example, the following **codo**-block defines a local operation for computing image contours:

```
contours :: CArray (Int, Int) Float → Float
contours = codo x ⇒ y ← gauss2D x
                    z ← gauss2D y
                    w ← (extract y) − (extract z)
                    laplace2D w
```

where *CArray i a* is a cursored-array data type, with index type i and element type a, and *gauss2D*, *laplace2D* :: *CArray* (*Int*, *Int*) *Float* → *Float* compute, at a particular index, *discrete Gaussian* and *Laplace* operators on 2D arrays. A contour image can thus be computed by applying (*extend contours*) to an image.

The primary contribution of this paper is the **codo**-notation, introduced in detail in Section 1, continuing with arrays as an example. The notation desugars into the operations of a comonad (Section 3) which provides an equational theory for the notation following from the laws of a comonad (Section 2). The **codo**-notation is analogous to the **do**-notation for programming with monads in Haskell, but with some notable differences which are explained from a *categorical semantics* perspective in Section 4. Section 5 discusses related work, including a comparison of the **codo**-notation to Haskell's *arrow notation*.

This paper contributes examples (arrays, trees, and graphs), explanation, and notation to promote comonads in programming. A prototype of the notation, as a macro-based library using quasi-quoting brackets, is provided by the `codo-notation` package.[2] An implementation as a GHC extension is in progress.

[2] `http://hackage.haskell.org/package/codo-notation`

Array example The array comonad is used throughout the next section to introduce **codo**. It is defined in Haskell by the following data type and instance:

$$\begin{array}{l}
\textbf{data}\ \mathit{CArray}\ i\ a = \mathit{CA}\ (\mathit{Array}\ i\ a)\ i \\
\textbf{instance}\ \mathit{Ix}\ i \Rightarrow \mathit{Comonad}\ (\mathit{CArray}\ i)\ \textbf{where} \\
\quad \mathit{extract}\ (\mathit{CA}\ a\ i) = a\,!\,i \\
\quad \mathit{extend}\,f\ (\mathit{CA}\ a\ i) = \textbf{let}\ es' = \mathit{map}\ (\lambda j \to (j, f\ (\mathit{CA}\ a\ j)))\ (\mathit{indices}\ a) \\
\qquad\qquad\qquad\qquad\quad \textbf{in}\ \ \mathit{CA}\ (\mathit{array}\ (\mathit{bounds}\ a)\ es')\ i
\end{array}$$

where *extract* accesses the cursor element using the array indexing operation !, and, for every index j of the parameter array, *extend* applies f to the array with j as its cursor, returning an index-value pair list from which the result array is constructed. Note, the return and parameter arrays have the same size and cursor, *i.e.*, *extend* preserves the incoming *context* in its result.

Many array operations can be defined as local operations $c\ a \to b$ (hereafter *comonadic operations*, sometimes called *coKleisli* arrows/morphisms in the literature) using relative indexing, *e.g.*, the *laplace2D* operator, for approximating differentiation, can be defined:

$$\begin{array}{l}
\mathit{laplace2D} :: \mathit{CArray}\ (\mathit{Int}, \mathit{Int})\ \mathit{Float} \to \mathit{Float} \\
\mathit{laplace2D}\ a = a\ ?\ (-1,0) + a\ ?\ (1,0) + a\ ?\ (0,-1) + a\ ?\ (0,1) - 4 * a\ ?\ (0,0)
\end{array}$$

where (?) abstracts relative indexing with bounds checking and default values:[3]

$$\begin{array}{l}
(?) :: (\mathit{Ix}\ i, \mathit{Num}\ a, \mathit{Num}\ i) \Rightarrow \mathit{CArray}\ i\ a \to i \to a \\
(\mathit{CA}\ a\ i)\ ?\ i' = \textbf{if}\ (\mathit{inRange}\ (\mathit{bounds}\ a)\ (i + i'))\ \textbf{then}\ a\,!\,(i + i')\ \textbf{else}\ 0
\end{array}$$

(where *Ix* is the class of valid array-index types). Whilst *laplace2D* computes the Laplacian at a single context (locally), *extend laplace2D* computes the Laplacian at every context (globally), returning an array rather than a single float.

1 Introducing *codo*

The **codo**-notation provides a form of *let*-binding for composing comonadic operations, which has the general form and type:

$$(\textbf{codo}\ p\ \Rightarrow\ \overline{p \leftarrow e};\ e) :: \mathit{Comonad}\ c \Rightarrow c\ t \to t'$$

(where p ranges over patterns, e over expressions, and t, t' over types). Compare this with the general form and type of the monadic **do**-notation:

$$(\textbf{do}\ \overline{p \leftarrow e}; e) :: \mathit{Monad}\ m \Rightarrow m\ t$$

Both comprise zero or more binding statements of the form $p \leftarrow e$ (separated by semicolons or new lines), preceding a final result expression. A **codo**-block however defines a function, with a pattern-match on its parameter following the **codo** keyword. The parameter is essential as comonads describe functions with structured input. A **do**-block is instead an expression (nullary function). Section 4 compares the two notations in detail.

[3] There are many alternative methods for abstracting boundary checking and values; our choice here is for simplicity of presentation rather than performance or accuracy.

Comonads and codo-notation for composition The *extend* operation of a comonad provides composition for comonadic functions as follows:

$$
\begin{aligned}
&(\hat{\circ}) :: \mathit{Comonad}\ c \Rightarrow (c\ y \to z) \to (c\ x \to y) \to c\ x \to z \\
&g\ \hat{\circ}\ f = g \circ (\mathit{extend}\ f)
\end{aligned} \tag{1}
$$

The laws of a comonad are equivalent to requiring that this composition is associative and that *extract* is its identity (discussed further in Section 2).

The **codo**-notation abstracts over *extend* in the composition of comonadic operations. For example, the composition of two array operations:

$$\mathit{lapGauss} = \mathit{laplace2D} \circ (\mathit{extend}\ \mathit{gauss2D})$$

($i.e.$, $\mathit{laplace2D}\ \hat{\circ}\ \mathit{gauss2D}$), can be written equivalently in the **codo**-notation:

$$
\begin{aligned}
\mathit{lapGauss} = \mathbf{codo}\ x \Rightarrow\ & y \leftarrow \mathit{gauss2D}\ x \\
& \mathit{laplace2D}\ y
\end{aligned}
$$

where $\mathit{lapGauss} :: \mathit{CArray}\ (\mathit{Int}, \mathit{Int})\ \mathit{Float} \to \mathit{Float}$, $x, y :: \mathit{CArray}\ (\mathit{Int}, \mathit{Int})\ \mathit{Float}$.

The parameter of a **codo**-block provides the context of the whole block where all subsequent local variables have the same context. For example, x and y in the above example block are arrays of the same size with the same cursor.

For a variable-pattern parameter, a **codo**-block is typed by the following rule: (here typing rules are presented with a single colon : for the typing relation)

$$[\text{varP}]\ \frac{\Gamma; x : c\ t \vdash_c e : t'}{\Gamma \vdash (\mathbf{codo}\ x \Rightarrow e) : \mathit{Comonad}\ c \Rightarrow c\ t \to t'}$$

where $\vdash_c$ types statements of a **codo**-block. Judgments $\Gamma; \Delta \vdash_c \ldots$ have two sequences of variable-type assumptions: Γ for variables outside a block and Δ for variables local to a block. For example, variable-pattern statements are typed:

$$[\text{varB}]\ \frac{\Gamma; \Delta \vdash_c e : t \qquad \Gamma; \Delta, x : c\ t \vdash_c r : t'}{\Gamma; \Delta \vdash_c x \leftarrow e; r : t'}$$

where r ranges over remaining statements and result expression $i.e.$ $r = \overline{p \leftarrow e}; e'$.

A variable-pattern statement therefore locally binds a variable, in scope for the rest of the block. The typing, where $e : t$ but $x : c\ t$, gives a hint about **codo** desugaring. Informally, ($\mathbf{codo}\ y \Rightarrow x \leftarrow e; e'$) is desugared into two functions, the first statement as $\lambda y \to e$ and the result expression as $\lambda x \to e'$. These are comonadically composed, $i.e.$, $(\lambda x \to e') \circ (\mathit{extend}\ (\lambda y \to e))$, thus $x : c\ t$.

Further typing rules for the **codo**-notation are collected in Fig. 1.

Non-linear plumbing For the *lapGauss* example, **codo** does not provide a significant simplification. The **codo**-notation more clearly benefits computations which are not mere linear function compositions. Consider a binary operation:

$$
\begin{aligned}
&\mathit{minus} :: (\mathit{Comonad}\ c, \mathit{Num}\ a) \Rightarrow c\ a \to c\ a \to a \\
&\mathit{minus}\ x\ y = \mathit{extract}\, x - \mathit{extract}\, y
\end{aligned}
$$

$$\boxed{\Gamma \vdash \textbf{codo}\ p \Rightarrow e : Comonad\ c \Rightarrow c\ t \to t'} \qquad \boxed{\Gamma; \Delta \vdash_c \overline{p \leftarrow e;} e : t}$$

$$[\text{varP}]\frac{\Gamma;\ x : c\ t \vdash_c\ e : t'}{\Gamma \vdash (\textbf{codo}\ x \Rightarrow e) : c\ t \to t'} \qquad [\text{varB}]\frac{\Gamma; \Delta \vdash_c\ e : t \quad \Gamma; \Delta, x : c\ t \vdash_c\ r : t'}{\Gamma; \Delta \vdash_c\ x \leftarrow e;\ r : t'}$$

$$[\text{tupP}]\frac{\Gamma;\ x : c\ t,\ y : c\ t' \vdash_c\ e : t''}{\Gamma \vdash (\textbf{codo}\ (x, y) \Rightarrow e) : c\ (t, t') \to t''} \qquad [\text{tupB}]\frac{\begin{array}{c}\Gamma; \Delta \vdash_c\ e : (t_1, t_2) \\ \Gamma; \Delta, x : c\ t_1, y : c\ t_2 \vdash_c\ r : t'\end{array}}{\Gamma; \Delta \vdash_c\ (x, y) \leftarrow e;\ r : t'}$$

$$[\text{wildP}]\frac{\Gamma; \cdot \vdash_c\ e : t}{\Gamma \vdash (\textbf{codo}\ \_ \Rightarrow e) : \forall a.c\ a \to t} \qquad [\text{letB}]\frac{\Gamma; \Delta \vdash_c\ e : t \quad \Gamma; \Delta, x : t \vdash_c\ r : t'}{\Gamma; \Delta \vdash_c\ \textbf{let}\ x = e;\ r : t'}$$

$$[\text{exp}]\frac{\Gamma \vdash e : t}{\Gamma; \cdot \vdash_c\ e : t} \quad [\text{var}]\frac{\Gamma, v : c\ t; \Delta \vdash_c\ e : t'}{\Gamma; \Delta, v : c\ t \vdash_c\ e : t'}$$

Fig. 1. Typing rules for the **codo**-notation

which subtracts its parameters at their respective current contexts. Using **codo**, *minus* can be used to compute a *pointwise* subtraction, *e.g.*

```
contours' = codo x ⇒ y ← gauss2D x
                     z ← gauss2D y
                     w ← minus y z
                     laplace2D w
```

(equivalent to *contours* in the introduction which inlined the definition of *minus*). The context, and therefore cursor, of every variable in the block is the same as that of x. Thus, y and z have the same cursor and *minus* is applied pointwise. The equivalent program without **codo** is considerably more complex:

```
contours' x = let y = extend gauss2D x
                  w = extend (λy' → let z = extend gauss2D y'
                                    in  minus y' z) y
              in laplace2D w
```

where the nested *extend* means that y' and z have the same cursor, thus *minus* y' z is pointwise. An alternate, more point-free, approach uses the composition $\hat{\circ}$:

$$contours' = laplace2D\ \hat{\circ}\ (\lambda y' \to minus\ y'\ \hat{\circ}\ gauss2D\ \$\ y')\ \hat{\circ}\ gauss2D$$

This approach resembles that of using monads without the do-notation, and is elegant for simple, linear function composition. However, for more complex plumbing the approach quickly becomes cumbersome. In the above two (non-**codo**) examples, care is needed to ensure that *minus* is applied pointwise. An incorrect attempt to simplify the first non-**codo** *contours'* might be:

```
contour_bad x = let y = extend gauss2D x
                    z = extend gauss2D y
                    w = extend (minus y) z
                in laplace2D w
```

In the above, *extend* (*minus* y) z subtracts z at every context from y at a particular, fixed context, *i.e.*, not a pointwise subtraction. An equivalent expression to *contours_bad* can be written using nested **codo**-blocks:

```
contour_bad = codo x ⇒ y ← gauss2D x
                       (codo y′ ⇒ z  ← gauss2D y′
                                  w ← minus y z
                                  laplace2D w) y
```

where y in *minus* y z is bound in the outer **codo**-block and thus has its cursor fixed, whilst z is bound in the inner **codo**-block and has its cursor varying. Variables bound outside of the nearest enclosing **codo**-block are "unsynchronised" with respect to the context inside the block, *i.e.*, at a different context.

A **codo**-block may have multiple parameters in an uncurried-style, via tuple patterns ([tupP], Fig. 1). For example, the following block has two parameters, which are Laplace-transformed and then pointwise added:

```
lapPlus :: CArray Int (Float, Float) → Float
lapPlus = codo (x, y) ⇒ a ← laplace2D x
                        b ← laplace2D y
                        (extract a) + (extract b)
```

This block has a single comonadic parameter with tuple elements, whose type is of the form $c\ (a, b)$. However, inside the block $x : c\ a$ and $y : c\ b$ as the desugaring of **codo** *unzips* the parameter (see Section 3). A comonadic tuple parameter ensures that multiple parameters have the same context, *e.g.*, x and y in the above example have the same shape/cursor. Therefore, a pair of arguments to *lapPlus* must be *zipped* first, provided by the *czip* operation:

```
class ComonadZip c where czip :: (c a, c b) → c (a, b)
```

For *CArray*, *czip* can be defined:

```
instance (Eq i, Ix i) ⇒ ComonadZip (CArray i) where
  czip (CA a i, CA a′ j) =
    if (i ≢ j ∨ bounds a ≢ bounds a′) then error "Shape/cursor mismatch"
    else let es″ = map (λk → (k, (a ! k, a′ ! k))) (indices a)
         in CA (array (bounds a) es″) i
```

Thus only arrays of the same shape and cursor can be zipped together. In the contextual understanding, the two parameter arrays are thus *synchronised* in their contexts. The example of *lapPlus* can be applied to two (synchronised) array parameters x and y by *extend lapPlus* (*czip* (x, y)).

Any data constructor pattern can be used for the parameter of a **codo**-block and on the left-hand side of a binding statement. For example, the following uses a tuple pattern in a binding statement (see [tupB], Fig. 1), which is equivalent to *lapPlus* by exchanging a parameter binding with a statement binding:

$$
\begin{aligned}
&\mathit{lapPlus} = \textbf{codo}\ z \Rightarrow (x, y) \leftarrow \mathit{extract}\ z \\
&\qquad\qquad\qquad\qquad a \leftarrow \mathit{laplace2D}\ x \\
&\qquad\qquad\qquad\qquad b \leftarrow \mathit{laplace2D}\ y \\
&\qquad\qquad\qquad\qquad (\mathit{extract}\ a) + (\mathit{extract}\ b)
\end{aligned}
$$

Tuple patterns are specifically discussed here since they provide multiple parameters to a **codo**-block, as seen above. The typing of a general pattern in a statement, for some type/data constructor T, is roughly as follows:

$$
[\text{patB}]\ \frac{\Gamma; \Delta \vdash_c e : T\,\bar{t} \qquad \Gamma; \Delta, \Delta' \vdash_c r : t' \qquad \mathit{dom}(\Delta') = \mathit{var\text{-}pats}(p)}{\Gamma; \Delta \vdash_c (T\,p) \leftarrow e;\ r : t'}
$$

where $\mathit{dom}(\Delta')$ is the set of variables in a sequence of typing assumptions, and *var-pats* is the set of variables occurring in a pattern.

Example: labelled graphs Many graph algorithms can be structured by a comonad, particularly compiler analyses and transformations on *control flow graphs* (CFGs). The following defines a labelled-graph comonad as a (non-empty) list of nodes which are pairs of a label and a list of their connected vertices:

```
data LGraph a = LG [(a, [Int])] Int   -- pre-condition: non-empty lists
instance Comonad LGraph where
  extract (LG xs c) = fst (xs !! c)
  extend f (LG xs c) = LG (map (λc' → (f (LG xs c'), snd (xs !! c')))
                                [0 .. length xs]) c
```

The *LGraph*-comonad resembles the array comonad where contexts are positions with a *cursor* denoting the current position. Analyses over CFGs can be defined using graphs labelled by syntax trees. For example, a *live-variable* analysis (which, for an imperative language, calculates the set of variables that may be used in a block before being (re)defined) can be written, using **codo**, as:

```
lva = codo g ⇒ lv0 ← (defUse g, [])   -- compute definition/use sets, paired
               lva' lv0                -- with initial empty live-variable set

lva' = codo ((def, use), lv) ⇒
            live_out ← foldl union [] (successors lv)
            live_in  ← union (extract def) ((extract live_out) \\ (extract use))
            lvp      ← ((extract def, extract use), extract live_in)
            lvNext   ← lva' lvp
            if (lv ≡ live_in) then (extract lv) else (extract lvNext)
```

where *union* and set difference (\\) on lists have type $Eq\ a \Rightarrow [a] \rightarrow [a] \rightarrow [a]$ and $\mathit{defUse} :: \mathit{LGraph}\ \mathit{AST} \rightarrow ([\mathit{Var}], [\mathit{Var}])$ computes the sets of variables defined and used by each block in a CFG. The analysis is recursive, refining the set of live variables until a fixed point is reached.

The live variables for every block of a CFG can be computed by *extend lva*.

Costate, trees, and zippers Arrays were used to introduce comonads and **codo** to aid understanding since the notion of *context* is made clear by the *cursor*. The above graph example has a similar form. Both are instances of a general comonad, often called the *costate* comonad, whose data type is a pair of a function from contexts to values and a particular context: $C\ a = (s \to a) \times s$.

For both arrays and labelled graphs, the type of contexts is a finite domain of integer, or integer-tuple, indices. For labelled graphs, the costate comonad is combined with *product comonad* (see [8]) pairing the label of a node with the list of its successors, thus the type is isomorphic to $C\ a = (s \to (a \times [s])) \times s$.

For *costate*, the notion of context is explicitly provided by a cursor acting as a *pointer* or *address*. This is not the only way to define a notion of context. Other data types encode the context structurally rather than using a cursor. For example, a comonad of labelled binary trees can be defined:

```
data BTree a = Leaf a | Node a (BTree a) (BTree a)
instance Comonad BTree where
  extract (Leaf a) = a
  extract (Node a l r) = a
  extend f (Leaf a) = Leaf (f (Leaf a))
  extend f t@(Node a l r) = Node (f t) (extend f l) (extend f r)
```

The action of *extend* is to apply its parameter function f to successive suffix trees, thus f can only access its children, not its parents. Thus *extend* not only defines what it means for a local (comonadic) operation to be applied globally, but also which contexts are *accessible* from each possible context.

A tree comonad that has a structural notion of context but whose comonadic operations can access any part of the tree can be defined using Huet's *zipper* data type, where trees are split into a path to the current position and the remaining parts of the tree [9, 5]. For a certain class of data types it has been shown that a zipper structure can be automatically derived by *differentiation* of the data type [10]. All container-like zippers are comonads [11] where the notion of context is encoded structurally, rather than by a *pointer*-like cursor. The **codo**-notation thus provides a convenient syntax for programming with zipper comonads.

2 Equational Theory

As shown in Section 1, *extend* provides composition for comonadic functions, Eq. (1). The laws of a comonad are exactly the laws that guarantee this composition is *associative* with *extract* as a *left* and *right unit*, *i.e.*

$$\begin{array}{rlcll}
\text{(right unit)} & f \mathbin{\hat{\circ}} \mathit{extract} \equiv f & \leadsto & \mathit{extend}\ \mathit{extract} \equiv \mathit{id} & \text{[C1]} \\
\text{(left unit)} & \mathit{extract} \mathbin{\hat{\circ}} f \equiv f & \leadsto & \mathit{extract} \circ (\mathit{extend}\ f) \equiv f & \text{[C2]} \\
\text{(associativity)} & h \mathbin{\hat{\circ}} (g \mathbin{\hat{\circ}} f) & \leadsto & \mathit{extend}\ g \circ \mathit{extend}\ f & \\
 & \quad \equiv (h \mathbin{\hat{\circ}} g) \mathbin{\hat{\circ}} f & & \quad \equiv \mathit{extend}\ (g \circ \mathit{extend}\ f) & \text{[C3]}
\end{array}$$

As there is no mechanism for enforcing such rules in Haskell the programmer is expected to verify the laws themselves.

Since **codo** is desugared into the operations of a comonad, the comonad laws imply equational laws for the **codo**-notation, shown in Fig. 2(a). Fig. 2(b) shows additional **codo** laws which follow from the desugaring.

Comonads are functors The category theoretic notion of a *functor* can be used to abstract *map*-like operations on parametric data types. In Haskell, functors are described by the *Functor* type class, of which *map* provides the list instance:

$$\textbf{class}\ \mathit{Functor}\ f\ \textbf{where}\ \mathit{fmap} :: (a \to b) \to f\ a \to f\ b$$
$$\textbf{instance}\ \mathit{Functor}\ [\,]\ \textbf{where}\ \mathit{fmap} = \mathit{map}$$

All comonads are functors by the following definition using *extend* and *extract*:

$$\mathit{cmap} :: \mathit{Comonad}\ c \Rightarrow (a \to b) \to c\ a \to c\ b$$
$$\mathit{cmap}\ f\ x = \mathit{extend}\ (f \circ \mathit{extract})$$

While *fmap* applies its parameter function to a single element of a data type, *extend* applies its parameter function to a subset (possibly the whole) of the parameter structure. Thus *extend* generalises *fmap*.

Monoidal operation The $\mathit{czip} :: (c\ a, c\ b) \to c\ (a, b)$ operation introduced in Section 1 corresponds to that of a *(semi)-monoidal functor* which may satisfy various laws with respect to the comonad (see the discussion of *(semi)-monoidal comonads* in [8]). The following property, which we call *idempotency* of a semi-monoidal functor, frequently holds of comonad/*czip* implementations:

$$\mathit{czip}\ (x, x) \equiv \mathit{cmap}\ (\lambda y \to (y, y))\ x \tag{2}$$

This property implies **codo** laws relating tuple patterns and *czip* (Fig. 2(c)). For every rule involving a tuple pattern there is an equivalent rule derived using the (χ) rule (Fig. 2(b)) which exchanges parameter and statement binders.

Shape preservation The *shape* of a data structure is defined by its structure without any values, which can be computed as such: (where $\mathit{const}\ x = \lambda\_ \to x$)

$$\mathit{shape} = \mathit{cmap}\ (\mathit{const}\ ())$$

An interesting derived property of comonads is that, for any comonadic function f, $(\mathit{extend}\ f)$ preserves the shape of the incoming structure in its result. For example, *extend* of the array comonad preserves the size, cursor, and dimensions of the parameter array in the result. Appendix A gives a proof of this property, which is stated formally, for a comonad c and function $f :: c\ a \to b$, as:

$$\mathit{shape} \circ (\mathit{extend}\, f) \equiv \mathit{shape} \tag{3}$$

This property explains why all locally bound variables in a **codo**-block bind comonadic values which have the same context.

(a) Comonad laws

[C1]
$$\begin{array}{ll} & \textbf{codo}\ x \Rightarrow f\ x \\ \equiv & \textbf{codo}\ x \Rightarrow y \leftarrow \mathit{extract}\ x \\ & \qquad\qquad f\ y \end{array}$$

[C2]
$$\begin{array}{ll} & \textbf{codo}\ x \Rightarrow f\ x \\ \equiv & \textbf{codo}\ x \Rightarrow y \leftarrow f\ x \\ & \qquad\qquad \mathit{extract}\ y \end{array}$$

[C3] (iff x is not free in e_1)
$$\begin{array}{ll} & \textbf{codo}\ x \Rightarrow y \leftarrow e_1 \\ & \qquad\qquad z \leftarrow e_2 \\ & \qquad\qquad e_3 \\ \equiv & \textbf{codo}\ x' \Rightarrow z \leftarrow (\textbf{codo}\ x \Rightarrow y \leftarrow e_1 \\ & \qquad\qquad\qquad\qquad\qquad e_2)\ x' \\ & \qquad\qquad e_3 \\ \equiv & \textbf{codo}\ x' \Rightarrow y \leftarrow e_1 \\ & \qquad\qquad (\textbf{codo}\ x \Rightarrow z \leftarrow e_2 \\ & \qquad\qquad\qquad\qquad e_3)\ x' \end{array}$$

(b) Pure laws

$$(\eta)\quad \textbf{codo}\ x \Rightarrow f\ x \equiv f$$

$$\begin{array}{ll} (\beta) & \textbf{codo}\ x \Rightarrow z \leftarrow (\textbf{codo}\ y \Rightarrow e_1)\ x \\ & \qquad\qquad e_2 \\ \equiv & \textbf{codo}\ x \Rightarrow y \leftarrow \mathit{extract}\ x \\ & \qquad\qquad z \leftarrow e_1 \\ & \qquad\qquad e_2 \end{array}$$

$$\begin{array}{ll} (\chi) & \textbf{codo}\ p \Rightarrow e \\ \equiv & \textbf{codo}\ z \Rightarrow p \leftarrow \mathit{extract}\ z \\ & \qquad\qquad e \end{array}$$

(c) Additional laws – if Eq. (2) holds

$$\begin{array}{ll} & \textbf{codo}\ x \Rightarrow f\ a\ b \\ \equiv & \textbf{codo}\ x \Rightarrow (a', b') \leftarrow \mathit{extract}\ (\mathit{czip}\ (a, b)) \\ & \qquad\qquad f\ a'\ b' \end{array}$$

$$\begin{array}{ll} & \textbf{codo}\ (b, c) \Rightarrow f\ (\mathit{czip}\ (b, c)) \\ \equiv & \textbf{codo}\ (b, c) \Rightarrow z \leftarrow (\mathit{extract}\ b, \mathit{extract}\ c) \\ & \qquad\qquad f\ z \end{array}$$

Fig. 2. Equational laws for the **codo**-notation

3 Desugaring *codo*

The desugaring of **codo** is based on Uustalu and Vene's semantics for a context-dependent λ-calculus [8]. It has two parts: translation of statements into composition via *extend*, and management of the environment for variables bound in a **codo**-block. The first part is explained by considering a restricted **codo**-notation, which only ever has one local variable, bound in the previous statement.

1). Single-variable environment For a comonad C, consider the **codo**-block:

$$\mathit{foo1} = (\textbf{codo}\ x \Rightarrow y \leftarrow f\ x;\ \ g\ y) :: C\ x \to z$$

where $f :: C\ x \to y$, $g :: C\ y \to z$. The first statement $y \leftarrow f\ x$ can be desugared as a function with parameter x and body $f\ x$, the second, which is the final result expression, can be similarly desugared as a function from y to its expression, *i.e.* $(\lambda x \to f\ x)$ and $(\lambda y \to g\ y)$. Both are functions with structured input, thus the semantics of *foo1* is their comonadic composition (equivalent to $g \mathbin{\hat{\circ}} f$):

$$[\![\mathit{foo1}]\!] = (\lambda y \to g\ y) \circ (\mathit{extend}\ (\lambda x \to f\ x)) :: Cx \to z.$$

2). Multiple-variable environment A **codo**-block may bind multiple variables, allowing the following example with binary function $h :: Cx \to Cy \to z$:

$$\mathit{foo2} = (\textbf{codo}\ x \Rightarrow y \leftarrow f\ x;\ \ h\ x\ y) :: C\ x \to z$$

The first statement cannot be desugared as before since the second statement uses both x and y, thus the desugaring must return x with the result of $f\ x$:

$$(\lambda x \to (\mathit{extract}\, x, f\ x)) :: C\ x \to (x, y) \qquad (\#4)$$

Applying *extract* to x means that *extend* (#4), of type $C\ x \to C\ (x, y)$, returns the parameter x and the result of $f\ x$ synchronised in their contexts.

The desugaring of the second statement is a function taking a value $C\ (x, y)$ and *unzipping* it, binding the constituent values to x and y in the scope of the result expression, where x and y are synchronised at the same context since *cmap* preserves the context encoded by the comonadic value:

$$\begin{aligned}(\lambda \mathit{env} \to \mathbf{let}\ x &= \mathit{cmap}\ \mathit{fst}\ \mathit{env}\\ y &= \mathit{cmap}\ \mathit{snd}\ \mathit{env}\ \mathbf{in}\ h\ x\ y) :: C\ (x, y) \to z \qquad (\#5)\end{aligned}$$

The desugaring of *foo2* is therefore $[\![\mathit{foo2}]\!] = (\#5) \circ (\mathit{extend}\, (\#4))$.

3.1 General construction

The desugaring translation traverses the list of binding statements in a **codo**-block, accumulating a comonadic environment of the local variables bound so far. The accumulated environment is structured by right-nested pairs terminated by a unit value (). Thus, the actual desugaring of *foo2* is:

$$\begin{aligned}[\![\mathit{foo2}]\!] = (&\lambda \mathit{env} \to \mathbf{let}\ y = \mathit{cmap}\ \mathit{fst}\ \mathit{env}\\ &\qquad\qquad\quad x = \mathit{cmap}\ (\mathit{fst} \circ \mathit{snd})\ \mathit{env}\ \mathbf{in}\ h\ x\ y)\\ &\circ (\mathit{extend}\, (\lambda \mathit{env} \to (\mathbf{let}\ x = \mathit{cmap}\ \mathit{fst}\ \mathit{env}\ \mathbf{in}\ f\ x, \mathit{extract}\ \mathit{env})))\\ &\circ (\mathit{cmap}\ (\lambda \mathit{env} \to (\mathit{env}, ())))\end{aligned}$$

For *foo2*, the environment in the first statement contains just x and has type $C\ (x, ())$, and in the second statement contains x and y and has type $C\ (y, (x, ()))$.

The top-level translation of a **codo**-block is defined:

$$\begin{aligned}[\![\mathbf{codo}\ x \Rightarrow b]\!] &= [\![x \vdash b]\!]_c \circ (\mathit{cmap}\ (\lambda x \to (x, ())))\\ [\![\mathbf{codo}\ \_ \Rightarrow b]\!] &= [\![\cdot \vdash b]\!]_c \circ (\mathit{cmap}\ (\lambda x \to (x, ())))\\ [\![\mathbf{codo}\ (x, y) \Rightarrow b]\!] &= [\![x, y \vdash b]\!]_c \circ \mathit{cmap}\ (\lambda p \to (\mathit{fst}\ p, (\mathit{snd}\ p, ())))\end{aligned}$$

where $[\![\Delta \vdash b]\!]_c$ is the translation of the binding statements b within a **codo**-block, with the scope of the local variables Δ. In the translation here, types are omitted for brevity. A translation with the types included can be found in the first author's forthcoming PhD dissertation [12].

The top-level translation generalises easily to arbitrary patterns. In each case, $[\![-]\!]_c$ is pre-composed with a lifted projection function, which projects values inside the incoming parameter comonad to right-nested pairs terminated by (). The translation of binding statements yields a Haskell function of type:

$$[\![\Delta \vdash \bar{b}\, ;\, e]\!]_c : \mathit{Comonad}\ c \Rightarrow c\, (t_1, (\ldots, (t_n, ()))) \to t$$

where $e : t$ and $\Delta = v_1, \ldots, v_n$ where $v_i : c\, t_i$. The definition of $[\![-]\!]_c$ is:

$$\begin{aligned}
[\![\Delta \vdash e]\!]_c &= [\![\Delta \vdash e]\!]_{exp} \\
[\![\Delta \vdash x \leftarrow e; r]\!]_c &= [\![x, \Delta \vdash r]\!]_c \circ \mathit{extend}\ (\lambda \mathit{env} \rightarrow ([\![\Delta \vdash e]\!]_{exp}\ \mathit{env}, \mathit{extract}\ \mathit{env})) \\
[\![\Delta \vdash (x, y) \leftarrow e; r]\!]_c &= [\![x, y, \Delta \vdash r]\!]_c \circ \mathit{extend}\ (\lambda \mathit{env} \rightarrow (\lambda((x, y), \Delta) \rightarrow (x, (y, \Delta))) \\
&\qquad\qquad ([\![\Delta \vdash e]\!]_{exp}\ \mathit{env}, \mathit{extract}\ \mathit{env}))
\end{aligned}$$

where $[\![\Delta \vdash e]\!]_{exp}$ translates expressions on the right-hand side of a statement or for the result of a block. The last case translates tuple-pattern statements where $\lambda((x, y), \Delta) \rightarrow (x, (y, \Delta)))$ reformats results into the right-nested tuple format of the environment; this generalises in the obvious way to arbitrary patterns.

The translation of expressions unzips the incoming comonadic environment, binding the values to the variables in Δ with a local *let*-binding:

$$[\![v_1, \ldots, v_n \vdash e]\!]_{exp} = \lambda \mathit{env} \rightarrow \textbf{let}\ [v_i = \mathit{cmap}\ (\mathit{fst} \circ \mathit{snd}^{i-1})\ \mathit{env}]_1^n\ \textbf{in}\ e$$

where snd^k means k compositions of *snd* and $\mathit{snd}^0 = \mathit{id}$.

The next section compares **codo**-notation with **do**-notation, and explains why the desugaring of **codo**-notation is more complex.

4 Comparing do- and codo-notation

Whilst comonads and monads are dual, this duality does not appear to extend to the **codo**- and **do**-notation. Both provide *let*-binding syntax, for composition of comonadic and monadic operations respectively. However, **codo**-blocks are parameterised, of type $c\ a \rightarrow b$ for a comonad c, whilst **do**-blocks are unparameterised, of type $m\ a$ for a monad m. Since comonads abstract functions with structured input, the parameter to a **codo**-block is important. In the **do**-notation, expressions have implicit input via their free variables and Haskell's scoping mechanism is reused for handling local variables in a **do**-block.

The **codo**- and **do**-notation can be seen as internal domain-specific languages, for contextual and effectful computations respectively, with their semantics defined by translation to Haskell. This perspective is similar to the approach of *categorical semantics*, where typed programs are given a denotation as a morphism[4] in some category, mapping from the inputs of a program to the outputs. The disparity between **codo**- and **do**-notation is illuminated by this approach.

Categorical semantics For the simply-typed λ-calculus, the traditional approach recursively maps the *type derivation* of an expression to a morphism [13]:

$$[\![\Gamma \vdash e : t]\!] : ([\![t_1]\!] \times \ldots \times [\![t_n]\!]) \longrightarrow [\![t]\!]$$

where $\Gamma = x_1 : t_1, \ldots x_n : t_n$. Thus, an expression $e : t$ with the free-variable typing assumptions Γ is modelled as a morphism from a *product* of the types for the free variables, as inputs, to the result type as the output.

[4] *Morphisms* generalise the notion of function. Readers unfamiliar with category theory may safely replace 'morphism' with 'function' here.

Categorical semantics for effectful computations Moggi showed that effectful computations can be given a semantics in terms of a *Kleisli* category [14, 15], which has morphisms $a \to m\ b$ for a monad m, with denotations:

$$[\![x_1 : t_1, \ldots x_n : t_n \vdash e : t]\!] : ([\![t_1]\!] \times \ldots \times [\![t_n]\!]) \longrightarrow m\ [\![t]\!]$$

In Moggi's calculus, *let*-binding corresponds to a call-by-value (eager) evaluation of effects followed by substitution of a pure value, corresponding to composition of the denotations provided by the *bind* operation of a monad. The semantics of multi-variable environments requires a *strong monad*: a monad with an additional *strength* operation. The effectful semantics for *let*-binding is as follows (here $a \xrightarrow{f} b$ abbreviates $f : a \to b$ with arrow concatenation expressing composition; $[\![-]\!]$ brackets are elided on types in morphisms for brevity):

$$\frac{[\![\Gamma \vdash e : t]\!] = g : \Gamma \to m\,t \qquad [\![\Gamma, x : t \vdash e' : t']\!] = h : \Gamma \times t \to m\,t'}{[\![\Gamma \vdash \mathbf{let}\ x = e\ \mathbf{in}\ e' : t']\!] = \Gamma \xrightarrow{\langle id, g\rangle} \Gamma \times m\,t \xrightarrow{strength} m(\Gamma \times t) \xrightarrow{bind\ h} m\,t'} \quad (6)$$

where $\langle f, g\rangle$ is the function pairing: $\lambda x \to (f\ x, g\ x)$, *bind* is the prefix version of Haskell's $(\gg=) :: Monad\ m \Rightarrow m\ a \to (a \to m\ b) \to m\ b$ operator and *strength* provides distributivity of $\times$ over m:

$$\begin{aligned} strength &: (a \times m\ b) \to m\ (a \times b) \\ bind &: (a \to m\ b) \to (m\ a \to m\ b) \end{aligned}$$

Whilst the **do**-notation provides a semantics for effectful *let*-binding embedded in Haskell, the translation is simplified by reusing Haskell's scoping mechanism since, in Haskell, all monads are *strong* with a canonical *strength*:

$$\begin{array}{l} strength :: Monad\ m \Rightarrow (a, m\ b) \to m\ (a, b) \\ strength\ (a, mb) = mb \gg= (\lambda b \to return\ (a, b)) \end{array}$$

It is straightforwardly proved that this definition of *strength* satisfies the properties of a *strong monad* (see [14] for these properties). The standard translation of **do** can be derived from (6) by inlining the above *strength* and simplifying according to the monad laws:

$$\frac{\Gamma \vdash e : m\,t \qquad \Gamma, x : t \vdash e' : m\,t'}{\Gamma \vdash [\![\mathbf{do}\ x \leftarrow e; e']\!] : m\,t' \ \equiv\ \Gamma \vdash e \gg= (\lambda x \to e') : m\,t'}$$

This gives a translation using just the monad operations and Haskell's scoping mechanism to define the semantics of multi-variable scopes for effectful *let*-binding. Thus the inputs to effectful computations are handled implicitly and so a **do**-block is an expression of type $m\ a$.

Categorical semantics for contextual computations The dual of Moggi's semantics interprets expressions in a *coKleisli category*, with denotations:

$$[\![x_1 : t_1, \ldots x_n : t_n \vdash e : t]\!] : c\,([\![t_1]\!] \times \ldots \times [\![t_n]\!]) \longrightarrow [\![t]\!]$$

for a comonad c. Uustalu and Vene gave the semantics of a *context-dependent* calculus in this form [8].

For a comonadic semantics, the input of an expression – the values of the free variables – thus have a *comonadic* product structure rather than just a product structure as in the monadic approach. Therefore, Haskell's scoping mechanisms cannot be directly used since the variables local to a **codo**-block must have the same comonadic context and are therefore wrapped in a comonadic data type. The local environment of a **codo**-block is therefore handled manually in the desugaring of **codo** resulting in a more complicated translation than that of the **do**-notation. The desugaring of statements is equivalent to the semantics of *let*-binding in Uustalu and Vene's approach:

$$\frac{[\![\Gamma \vdash e : t]\!] = g : c\,\Gamma \to\ t \qquad [\![\Gamma, x : t \vdash e' : t']\!] = h : c\,(\Gamma \times t) \to t'}{[\![\Gamma \vdash \mathbf{let}\ x =\ e\ \mathbf{in}\ e' : t']\!] = c\ \Gamma \xrightarrow{extend\langle extract, g\rangle} c\,(\Gamma \times t) \xrightarrow{h} t'}$$

The other parts of the desugaring manage projections from the (comonadic) environment, simulating application and variable access in a comonadic semantics.

5 Related Work and Conclusions

Arrow notation In Haskell, various notions of computation can be encoded as a *category* structure, with additional *arrow* operations for constructing computations and handling environments, defined by the *Category* and *Arrow* classes:

```
class Category cat where                  class Category a ⇒ Arrow a where
  id :: cat x x                             arr :: (x → y) → a x y
  (∘) :: cat y z → cat x y → cat x z        first :: a x y → a (x, z) (y, z)
```

A *Category* thus has a notion of composition and identity for its morphisms, which are modelled by the type *cat x y*. The *Arrow* class provides *arr* for promoting a Haskell function to a morphism and *first* transforms a morphism to take and return an extra parameter, used for threading an environment through a computation. Other arrow combinators can be derived from this minimal set.

Every comonad defines a *coKleisli category*, whose morphisms have structured input, where composition is defined as in Section 1. Furthermore, all coKleisli categories in Haskell are *arrows*:

```
data CoKleisli c x y = CoK {unCoK :: (c x → y)}
instance Comonad c ⇒ Category (CoKleisli c) where
  id = CoK extract
  (CoK g) ∘ (CoK f) = CoK (g ∘ (extend f))
instance Comonad c ⇒ Arrow (CoKleisli c) where
  arr k = CoK (k ∘ extract)
  first (CoK f) = CoK (λx → (f (cmap fst x), extract (cmap snd x)))
```

where *arr* pre-composes a function with *extract*, and *first* is defined similarly to the handling of the local block environment in the desugaring of **codo**.

The *arrow notation* simplifies programming with arrows [16, 17], comprising: *arrow formation* (**proc** $x \to e$), *arrow application* ($f \prec x$) and *binding* ($x \leftarrow e$). Given the above coKleisli instances for *Category* and *Arrow*, comonadic operations can be written in the arrow notation instead of using the **codo**-notation. For example, the original *contours* example can be written as follows:

```
proc x → do y ← CoK gauss2D ≺ x
            z ← CoK gauss2D ≺ y
            w ← returnA ≺ y − z
            CoK laplace2D ≺ w
```

The *arrow* notation here is not much more complicated than **codo**, requiring just the additional overhead of the arrow application operator $\prec$ and lifting of *gauss2D* and *laplace2D* by *CoK*. One difference is that the variables here have a non-comonadic type, *i.e.*, *Float* rather than *CArray* (*Int*, *Int*) *Float*.

The arrow notation is however more cumbersome than **codo** when plumbing comonadic values, for example when using comonadic binary functions (of type $c\ t \to c\ t' \to t''$). The alternate definition of *contours* using *minus* becomes:

```
proc x → do y ← CoK gauss2D ≺ x
            z ← CoK gauss2D ≺ y
            w ← CoK (λv → minus (fmap fst v) (fmap snd v)) ≺ (y, z)
            CoK laplace2D ≺ w
```

where $v :: c\ (y, z)$ must be deconstructed manually. Whilst *minus* can be inlined here and the code rewritten to the more elegant first example, this is only possible since *minus* applies *extract* to both arguments. For other comonadic operations, with more complex behaviour, this refactoring is not always possible.

Comparing the two, *arrow* notation appears as powerful as **codo**-notation, in terms of the computations which can be expressed. Indeed, from a categorical perspective, both notations need only a comonad structure (*i.e.*, coKleisli category) with no additional *closed* or *monoidal* structure (see Paterson's discussion [17, §2.1]). However, whilst arrow notation is almost as simple as **codo** for some purposes, the syntax is less natural for more complicated plumbing of comonadic values (as seen above). We argue that **codo** provides the most elegant and natural solution to programming with comonads, with a cleaner applicative-style.

Other applications There are many interesting comonads which have not been explored here. For example, the semantics of the Lucid dataflow language are captured by an *infinite stream comonad* [4], which was used by Uustalu and Vene to define an interpreter for Lucid in Haskell. Using **codo**-notation, Lucid can be embedded directly into Haskell as an internal domain-specific language.

Many comonadic data types are instances of the general concept of *containers*. *Containers* comprise a set of *shapes* S and, for each shape $s \in S$, a type of *positions* Ps, with the data type $C\,a = \sum_{s \in S}(Ps \to a)$, *i.e.*, a coproduct of functions from positions to values for each possible shape [18]. Ahman *et al.* recently showed that all *directed containers* (those with notions of sub-shape) are comonads, where positions are contexts and sub-shapes define accessibility between contexts for the definition of *extend* [11]. The labelled binary-tree example in Section 1 can be described as a directed-container comonad. The *costate* comonad can be generalised to *cursored containers* with type $C\,a = \sum_{s \in S}(P s \rightarrow a) \times P s$.

Whilst the **codo**-notation was developed here in Haskell, it could be applied in other languages with further benefits. For example, a **codo**-notation for ML could be used to abstract laziness using a *delayed-computation* comonad with data type $C\ a = () \rightarrow a$, or defining lazy lists using the stream comonad.

Concluding remarks Comonads essentially abstract boilerplate code for data structure traversals, allowing succinct definitions of *local* operations by abstracting their promotion to *global* operations. The **codo**-notation presented here simplifies programming with comonads. We hope this prompts the use of comonads as a design pattern and tool for abstraction, and promotes further exploration of comonads yielding new and interesting examples.

Whilst the **codo** keyword is used in the notation here, some may prefer an alternate keyword as **codo**-notation is not exactly dual to **do**-notation (Section 4). For example, using **context** as the keyword provides more intuition about its use, akin to **do**, but causes more serious namespace pollution.

Acknowledgements We thank Jeremy Gibbons, Ralf Hinze, Tomas Petricek, Tarmo Uustalu, and Varmo Vene for helpful discussions, and to the anonymous reviewers for their comments on this paper and an earlier draft. This research was supported by an EPSRC Doctoral Training Award.

References

1. Wadler, P.: The essence of functional programming. In: Proceedings of POPL '92, ACM (1992) 1–14
2. Wadler, P.: Monads for functional programming. Advanced Functional Programming (1995) 24–52
3. Petricek, T., Syme, D.: Syntax Matters: writing abstract computations in F#. In: Pre-proceedings of TFP (Trends in Functional Programming), St. Andrews, Scotland (2012)
4. Uustalu, T., Vene, V.: The Essence of Dataflow Programming. Lecture Notes in Computer Science **4164** (Nov 2006) 135–167
5. Uustalu, T., Vene, V.: Comonadic functional attribute evaluation. Trends in Functional Programming-Volume 6 (2007) 145–160
6. Orchard, D., Bolingbroke, M., Mycroft, A.: Ypnos: declarative, parallel structured grid programming. In: DAMP '10, NY, USA, ACM (2010) 15–24
7. Kieburtz, R.B.: Codata and Comonads in Haskell (1999)
8. Uustalu, T., Vene, V.: Comonadic Notions of Computation. Electron. Notes Theor. Comput. Sci. **203**(5) (2008) 263–284
9. Huet, G.: The zipper. Journal of Functional Programming **7**(05) (1997) 549–554
10. McBride, C.: The derivative of a regular type is its type of one-hole contexts. Unpublished manuscript (2001)

11. Ahman, D., Chapman, J., Uustalu, T.: When is a container a comonad? Foundations of Software Science and Computational Structures (2012) 74–88
12. Orchard, D.: Programming contextual computations (2013) Forthcoming PhD dissertation, `http://www.cl.cam.ac.uk/techreports`.
13. Lambek, J., Scott, P.: Introduction to higher-order categorical logic. Cambridge University Press (1988)
14. Moggi, E.: Computational lambda-calculus and monads. In: Logic in Computer Science, 1989. LICS'89, IEEE (1989) 14–23
15. Moggi, E.: Notions of computation and monads. Information and computation **93**(1) (1991) 55–92
16. Hughes, J.: Programming with arrows. Advanced Functional Programming (2005) 73–129
17. Paterson, R.: A new notation for arrows. In: ACM SIGPLAN Notices. Volume 36., ACM (2001) 229–240
18. Abbott, M., Altenkirch, T., Ghani, N.: Containers: constructing strictly positive types. Theoretical Computer Science **342**(1) (2005) 3–27

A Proof of shape preservation

To prove shape preservation we first prove the following intermediate lemma:

$$\mathit{cmap}\ g \circ \mathit{extend}\, f = \mathit{extend}\, (g \circ f) \tag{7}$$

$$\begin{array}{ll}
& \mathit{cmap}\ g \circ \mathit{extend}\, f \\
\equiv & \mathit{extend}\, (g \circ \mathit{extract}) \circ \mathit{extend}\, f \quad & \text{definition of } \mathit{cmap} \\
\equiv & \mathit{extend}\, (g \circ \mathit{extract} \circ \mathit{extend}\, f) & [\text{C3}] \\
\equiv & \mathit{extend}\, (g \circ f) \quad \square & [\text{C2}]
\end{array}$$

The proof of shape preservation (3) is then:

$$\begin{array}{ll}
& \mathit{shape} \circ (\mathit{extend}\, f) \\
\equiv & (\mathit{cmap}\ (\mathit{const}\ ())) \circ (\mathit{extend}\, f) & \text{definition of } \mathit{shape} \\
\equiv & \mathit{extend}\, ((\mathit{const}\ ()) \circ f) & (7) \\
\equiv & \mathit{extend}\, ((\mathit{const}\ ()) \circ \mathit{extract}) & (\mathit{const}\ x) \circ f \equiv (\mathit{const}\ x) \circ g \\
\equiv & (\mathit{cmap}\ (\mathit{const}\ ())) \circ (\mathit{extend}\ \mathit{extract}) & (7) \\
\equiv & \mathit{cmap}\ (\mathit{const}\ ()) & [\text{C1}] \\
\equiv & \mathit{shape} & \text{definition of } \mathit{shape}
\end{array}$$

$\square$