Chapter 10

Comonadic functional attribute evaluation

Tarmo Uustalu[1] and Varmo Vene[2]

Abstract: We have previously demonstrated that dataflow computation is comonadic. Here we argue that attribute evaluation has a lot in common with dataflow computation and admits a similar analysis. We claim that this yields a new, modular way to organize both attribute evaluation programs written directly in a functional language as well as attribute grammar processors. This is analogous to the monadic approach to effects. In particular, we advocate it as a technology of executable specification, not as one of efficient implementation.

10.1 INTRODUCTION

Following on from the seminal works of Moggi [Mog91] and Wadler [Wad92], monads have become a standard tool in functional programming for structuring effectful computations that are used both directly in programming and in language processors. In order to be able to go also beyond what is covered by monads, Power and Robinson [PR97] invented the concept of Freyd categories. Hughes [Hug00] proposed the same, unaware of their work, under the name of arrow types. The showcase application example of Freyd categories/arrow types has been dataflow computation, which, for us, is an umbrella name for various forms of computation based on streams or timed versions thereof and characterizes, in particular, languages like Lucid [AW85], Lustre [HCRP91] and Lucid Synchrone [CP96].

In two recent papers [UV05a, UV05b], we argued that, as far as dataflow computation is concerned, a viable alternative to Freyd categories is provided by

[1]Inst. of Cybernetics at Tallinn Univ. of Technology, Akadeemia tee 21, EE-12618 Tallinn, Estonia; Email `tarmo@cs.ioc.ee`

[2]Dept. of Computer Science, Univ. of Tartu, J. Liivi 2, EE-50409 Tartu, Estonia; Email `varmo@cs.ut.ee`

something considerably more basic and standard, namely comonads, the formal dual of monads. In fact, comonads are even better, as they explicate more of the structure present in dataflow computations than the laxer Freyd categories. Comonads in general should be suitable to describe computations that depend on an implicit context. Stream functions as abstractions of transformers of discrete-time signals turn out to be a perfect example of such computations: the value of the result stream in a position of interest (the present of the result signal) may depend not only on the value in the argument stream in the same position (the present of the argument signal), but also on other values in it (its past or future or both). We showed that general, causal and anticausal stream functions are described by comonads and that explicit use of the appropriate comonad modularizes both stream-based programs written in a functional language and processors of stream-based languages.

In this paper, we demonstrate that attribute evaluation from attribute grammars admits a similar comonadic analysis. In attribute grammars [Knu68], the value of an attribute at a given node in a syntax tree is defined by the values of other attributes at this and other nodes. Also, an attribute definition only makes sense relative to a suitable node in a tree, but nodes are never referenced explicitly in such definitions: context access happens solely via operations for relative local navigation. This hints that attribute grammars exhibit a form of dependence on an implicit context which is quite similar to that present in dataflow programming. We establish that this form of context-dependence is comonadic and discuss the implications. In particular, we obtain a new, modular way to organize attribute evaluation programs, which is radically different from the approaches that only use the initial-algebraic structure of tree types.

Similarly to the monadic approach to effects, this is primarily to be seen as an executable specification approach. As implementations, our evaluators will normally be grossly inefficient, unless specifically fine-tuned, but as specifications, they are of a very good format: they are concise and, because of their per-attribute organization, smoothly composable (in the dimension of composing several attribute grammars over the same underlying context-free grammar). Systematic transformation of these reference evaluators into efficient implementations ought to be possible, we conjecture, but this is a different line of work. This is highly analogous to the problem of efficient compilation of dataflow programs (not very hard in the case of causal languages, but a real challenge in the case of languages that support anticipation).

We are not aware of earlier comonadic or arrow-based accounts of attribute evaluation. But functional attribute evaluation has been a topic of research for nearly 20 years, following on from the work of Johnsson [Joh87]. Some of this work, concerned with per-attribute compositional specification of attribute grammars, is mentioned in the related work section below.

The paper is organized as follows. In Section 10.2, we overview our comonadic approach to dataflow computation and processing of dataflow languages. In Section 10.3, we demonstrate that the attribute evaluation paradigm can also be analyzed comonadically, studying separately the simpler special case of purely

synthesized attributed grammars and the general case. We also provide a discussion of the strengths and problems with the approach. Section 10.4 is a brief review of the related work whereas Section 10.5 summarizes. Most of the development is a carried out in the programming language Haskell, directly demonstrating that the approach is implementable (on the level of executable specifications). The code is Haskell 98 extended with multiparameter classes.

10.2 COMONADS AND DATAFLOW COMPUTATION

We begin by mentioning the basics about comonads to then quickly continue with a dense review of comonadic dataflow computation [UV05a, UV05b].

Comonads are the formal dual of monads, so a *comonad* on a category $\mathcal{C}$ is given by a mapping $D : |\mathcal{C}| \to |\mathcal{C}|$ (by $|\mathcal{C}|$ we mean the collection of the objects of $\mathcal{C}$) together with a $|\mathcal{C}|$-indexed family ε of maps $\varepsilon_A : DA \to A$ (*counit*), and an operation $-^\dagger$ taking every map $k : DA \to B$ in $\mathcal{C}$ to a map $k^\dagger : DA \to DB$ (*coextension operation*) such that

1. for any $k : DA \to B$, $\varepsilon_B \circ k^\dagger = k$,
2. ${\varepsilon_A}^\dagger = \mathsf{id}_{DA}$,
3. for any $k : DA \to B$, $\ell : DB \to C$, $(\ell \circ k^\dagger)^\dagger = \ell^\dagger \circ k^\dagger$.

Any comonad $(D, \varepsilon, -^\dagger)$ defines a category $\mathcal{C}_D$ where $|\mathcal{C}_D| = |\mathcal{C}|$ and $\mathcal{C}_D(A,B) = \mathcal{C}(DA,B)$, $(\mathsf{id}_D)_A = \varepsilon_A$, $\ell \circ_D k = \ell \circ k^\dagger$ (*coKleisli category*) and an identity on objects functor $J : \mathcal{C} \to \mathcal{C}_D$ where $Jf = f \circ \varepsilon_A$ for $f : A \to B$.

CoKleisli categories make comonads relevant for analyzing notions of context-dependent function. If the object DA is viewed as the type of contextually situated values of A, a context-dependent function from A to B is a map $DA \to B$ in the base category, i.e., a map from A to B in the coKleisli category. The counit $\varepsilon_A : DA \to A$ discards the context of its input whereas the coextension $k^\dagger : DA \to DB$ of a function $k : DA \to B$ essentially duplicates it (to feed it to k and still have a copy left). The function $Jf : DA \to B$ is a trivially context-dependent version of a pure function $f : A \to B$.

In Haskell, we can define comonads as a type constructor class.

```
class Comonad d where
  counit :: d a -> a
  cobind :: (d a -> b) -> d a -> d b
```

Some examples are the following:

- $DA = A$, the identity comonad,
- $DA = A \times E$, the product comonad,
- $DA = \mathsf{Str}A = \nu X.A \times X$, the streams comonad ($\nu$ denoting the greatest fixed-point operator)

The stream comonad Str is defined as follows:

```
data Stream a = a :< Stream a                    -- coinductive

instance Comonad Stream where
  counit (a :< _) = a
  cobind k d@(_ :< as) = k d :< cobind k as
```

(Note that we denote the cons constructor of streams by $:<$.)

This comonad is the simplest one relevant for dataflow computation. Intuitively, it is the comonad of future. In a value of type $\mathsf{Str}A \cong A \times \mathsf{Str}A$, the first component of type A is the main value of interest while the second component of type $\mathsf{Str}A$ is its context. In our application, the first is the present and the second is the future of an A-signal. The coKleisli arrows $\mathsf{Str}A \to B$ represent those functions $\mathsf{Str}A \to \mathsf{Str}B$ that are anticausal in the sense only the present and future of an input signal can influence the present of the output signal. The interpretation of these representations as stream functions is directly provided by the coextension operation:

```
class SF d where
  run :: (d a -> b) -> Stream a -> Stream b

instance SF Stream where
  run k = cobind k
```

A very important anticausal function is unit anticipation (cf. the 'next' operator of dataflow languages):

```
class Antic d where
  next :: d a -> a

instance Antic Stream where
  next (_ :< (a' :< _)) = a'
```

To be able to represent general stream functions, where the present of the output can depend also on the past of the input, we must employ a different comonad LS. It is defined by $\mathsf{LS}A = \mathsf{List}A \times \mathsf{Str}A$ where $\mathsf{List}A = \mu X.1 + X \times A$ is the type of (snoc-)lists over A (μ denoting the least fixedpoint operator). The idea is that a value of $\mathsf{LS}A \cong \mathsf{List}A \times (A \times \mathsf{Str}A)$ can record the past, present and future of a signal. (Notice that while the future of a signal is a stream, the past is a list: it must be finite.) Note that, alternatively, we could have defined $\mathsf{LS}A = \mathsf{Str}A \times \mathsf{Nat}$ (a value in a context is the entire history of a signal together with a distinguished time instant). This comonad is Haskell-defined as follows. (Although in Haskell there is no difference between inductive and coinductive types, in the world of sets and functions the definition of lists below should be read inductively.)

```
data List a = Nil | List a :> a                  -- inductive
```

```
data LS  a = List a :=| Stream a

instance Comonad LS where
  counit (_ :=| (a :< _)) = a
  cobind k d = cobindL k d :=| cobindS k d
   where cobindL k (Nil     :=| _ ) = Nil
         cobindL k (az :> a :=| as) = cobindL k d' :> k d'
                                where d' = az :=| (a :< as)
         cobindS k d@(az :=| (a :< as)) = k d :< cobindS k d'
                                where d' = az :> a :=| as
```

(We denote the snoc constructor of lists by $:>$ and pairing of a list and a stream by $:=|$. Now the visual purpose of the notation becomes clear: in values of type $\mathsf{LS}\ A$, both the snoc constructors of the list (the past of a signal) and the cons constructors of the stream (the present and future) point to the present which follows the special marker $:=|$.)

The interpretation of coKleisli arrows as stream functions and the representation of unit anticipation are defined as follows:

```
instance SF LS where
  run k as = bs where (Nil :=| bs) = cobind k (Nil :=| as)

instance Antic LS where
  next (_ :=| (_ :< (a' :< _))) = a'
```

With the LS comonad it is possible to represent also the important parameterized causal function of initialized unit delay (the 'followed-by' operator):

```
class Delay d where
  fby :: a -> d a -> a

instance Delay LS where
  a0 `fby` (Nil     :=| _) = a0
  _  `fby` (_ :> a' :=| _) = a'
```

Relevantly for "physically" motivated dataflow languages (where computations input or output physical dataflows), it is also possible to characterize causal stream functions as a coKleisli category. The comonad LV is defined by $\mathsf{LV}A = \mathsf{List}A \times A$, which is obtained from $\mathsf{LS}A \cong \mathsf{List}A \times (A \times \mathsf{Str}A)$ by removing the factor of future. This comonad is Haskell-implemented as follows.

```
data LV a = List a := a

instance Comonad LV where
  counit (_ := a) = a
  cobind k d@(az := _) = cobindL k az := k d
     where cobindL k Nil       = Nil
           cobindL k (az :> a) = cobindL k az :> k (az := a)
```

```
instance SF LV where
  run k as = run' k (Nil :=| as)
    where run' k (az :=| (a :< as))
                   = k (az := a) :< run' k (az :> a :=| as)

instance Delay LV where
  a0 `fby` (Nil       := _) = a0
  _  `fby` ((_ :> a') := _) = a'
```

Various stream functions are beautifully defined in terms of comonad operations and the additional operations of anticipation and delay. Some simple examples are the Fibonacci sequence, summation and averaging over the immediate past, present and immediate future:

```
fib :: (Comonad d, Delay d) => d () -> Integer
fib d = 0 `fby`
            cobind (\ e -> fib e + (1 `fby` cobind fib e)) d

sum :: (Comonad d, Delay d) => d Integer -> Integer
sum d = (0 `fby` cobind sum d) + counit d

avg :: (Comonad d, Antic d, Delay d) => d Integer -> Integer
avg d = ((0 `fby` d) + counit d + next d) `div` 3
```

In a dataflow language, we would write these definitions like this.

$$\begin{aligned} \mathit{fib} &= 0 \text{ fby } (\mathit{fib} + (1 \text{ fby } \mathit{fib})) \\ \mathit{sum}\ x &= (0 \text{ fby } \mathit{sum}\ x) + x \\ \mathit{avg}\ x &= ((0 \text{ fby } x) + x + \text{next } x)/3 \end{aligned}$$

In [UV05b], we also discussed comonadic processors of dataflow languages, in particular the meaning of higher-order dataflow computation (the interpretation of lambda-abstraction); for space reasons, we cannot review this material here.

10.3 COMONADIC ATTRIBUTE EVALUATION

We are now ready to describe our comonadic approach to attribute evaluation. Attribute evaluation is similar to stream-based computation in the sense that there is a fixed (skeleton of a) datastructure on which computations are done. We will build on this similarity.

10.3.1 Attribute grammars

An attribute grammar as a specification of an annotation (attribution) of a syntax tree [Knu68] is a construction on top of a context-free grammar. To keep the presentation simple and to circumvent the insufficient expressiveness of Haskell's

type system (to be discussed in Sec. 10.3.5), we consider here a fixed context-free grammar with a single nonterminal S with two associated production rules

$$\begin{array}{lcl} S & \longrightarrow & E \\ S & \longrightarrow & SS \end{array}$$

where E is a pseudo-nonterminal standing for some set of terminals.

An attribute grammar extends its underlying context-free grammar with attributes and semantic equations. These are attached to the nonterminals and the production rules of the context-free grammar. A semantic equation determines the value of an attribute at a node in a production rule application via the values of other attributes at other nodes involved. We explain this on examples. Let us use superscripts ℓ, b and subscripts L, R as a notational device to tell apart the different occurrences of the nonterminal S in the two production rules as follows:

$$\begin{array}{lcl} S^{\ell} & \longrightarrow & E \\ S^{b} & \longrightarrow & S^{b}_{L} S^{b}_{R} \end{array}$$

Now we can, for example, equip the nonterminal S with two boolean attributes *avl*, *locavl* and a natural-number attribute *height* and govern them by semantic equations

$$\begin{array}{rcl} S^{\ell}.avl & = & \mathrm{tt} \\ S^{b}.avl & = & S^{b}_{L}.avl \wedge S^{b}_{R}.avl \wedge S^{b}.locavl \\ S^{\ell}.locavl & = & \mathrm{tt} \\ S^{b}.locavl & = & |S^{b}_{L}.height - S^{b}_{R}.height| \leq 1 \\ S^{\ell}.height & = & 0 \\ S^{b}.height & = & \max(S^{b}_{L}.height, S^{b}_{R}.height) + 1 \end{array}$$

This gives us an attribute grammar for checking if an S-tree (a syntax tree whose root is an S-node) is AVL.

We can also, for example, equip the nonterminal S with natural-number attributes *numin*, *numout* and subject them to equations

$$\begin{array}{rcl} S^{b}_{L}.numin & = & S^{b}.numin + 1 \\ S^{b}_{R}.numin & = & S^{b}_{L}.numout + 1 \\ S^{\ell}.numout & = & S^{\ell}.numin \\ S^{b}.numout & = & S^{b}_{R}.numout \end{array}$$

This is a grammar for pre-order numbering of the nodes of a tree. The attribute *numin* corresponds to the pre-order numbering, the attribute *numout* is auxiliary.

We can see that the value of an attribute at a node can depend on the values of that and other attributes at that node and the children nodes (as in the case of *avl*, *locavl*, *height*, *numout*) or on the values of that and other attributes at that

node, the parent node and sibling nodes (*numin*). Attributes of the first kind are called synthesized. Attributes of the second kind are called inherited. Attribute grammars are classified into purely synthesized attribute grammars and general attribute grammars (where there are also inherited attributes).

The problem of attribute evaluation is to compute the full attribution of a given grammatical tree (given the values of the inherited attributes at the root), but one may of course really care about selected attributes of selected nodes. E.g., in the case of AVLness, we are mostly interested in the value of *avl* at the root, while, in the case of pre-order numbering, our concern is the attribute *numin*.

The type of attributed grammatical S-trees is

$$\begin{aligned} \mathsf{Tree}\,E\,A &= \mu X.\,A \times (E + X \times X) \\ &\cong A \times (E + \mathsf{Tree}\,E\,A \times \mathsf{Tree}\,E\,A) \end{aligned}$$

where A is the type of S-attributes of interest (aggregated into records). In Haskell, we can define:

```
data Tree e a = a :< Trunk e (Tree e a)

data Trunk e x = Leaf e | Bin x x
```

(Now $:<$ is a constructor for making an attributed tree.)

An attribute evaluator in the conventional sense is a tree transformer of type $\mathsf{Tree}\,E\,1 \to \mathsf{Tree}\,E\,A$ where A is the type of records of all S-attributes of the grammar.

10.3.2 Comonadic purely synthesized attributed grammars

In the case of a purely synthesized attribute grammar, the local value of the defined attribute of an equation can only depend on the local and children-node values of the defining attributes. This is similar to anticausal stream-computation. The relevant comonad is the comonad structure on $\mathsf{Tree}\,E$. The idea that the second component of a value in $\mathsf{Tree}\,E\,A \cong A \times (E + \mathsf{Tree}\,E\,A \times \mathsf{Tree}\,E\,A)$ (the terminal at a leaf or the subtrees rooted by the children of a binary node) is obviously the natural datastructure to record the course of an attribute below a current node and in a purely synthesized grammar the local value of an attribute can only depend on the values of that and other attributes at the given node and below. The comonad structure is Haskell-defined as follows, completely analogously to the comonad structure on Str.

```
instance Comonad (Tree e) where
  counit (a :< _) = a
  cobind k d@(_ :< as) = k d :< case as of
            Leaf e      -> Leaf e
            Bin asL asR -> Bin (cobind k asL) (cobind k asR)
```

The coKleisli arrows of the comonad are interpreted as tree functions by the coextension operation as in the case of Str. Looking up the attribute values at

the children of a node (which is needed to define the local values of synthesized attributes) can be done via an operation similar to 'next'.

```
class TF e d where
  run :: (d e a -> b) -> Tree e a -> Tree e b

instance TF e Tree where
  run = cobind

class Synth e d where
  children :: d e a -> Trunk e a

instance Synth e Tree where
  children (_ :< as) = case as of
                         Leaf e                    -> Leaf e
                         Bin (aL :< _) (aR :< _) -> Bin aL aR
```

10.3.3 Comonadic general attributed grammars

To be able to define attribute evaluation for grammars that also have inherited attributes (so the local value of an attribute can be defined through the values of other attributes at the parent or sibling nodes), one needs a notion of context that can store also store the upper-and-surrounding course of an attribute. This is provided by Huet's generic zipper datastructure [Hue97], instantiated for our tree type constructor. The course of an attribute above and around a given node lives in the type

$$\begin{aligned} \mathsf{Path}\,E\,A &= \mu X.\,1 + X \times (A \times \mathsf{Tree}\,E\,A + A \times \mathsf{Tree}\,E\,A) \\ &\cong 1 + \mathsf{Path}\,E\,A \times (A \times \mathsf{Tree}\,E\,A + A \times \mathsf{Tree}\,E\,A) \end{aligned}$$

of *path structures*, which are snoc-lists collecting the values of the attribute at the nodes on the path up to the root and in the side subtrees rooted by these nodes. A *zipper* consists of a tree and a path structure, which are the subtree rooted by a node and the path structure up to the root of the global tree, and records both the local value and lower and upper-and-surrounding courses of an attribute: we define

$$\begin{aligned} \mathsf{Zipper}\,E\,A &= \mathsf{Path}\,E\,A \times \mathsf{Tree}\,E\,A \\ &\cong \mathsf{Path}\,E\,A \times (A \times (E + \mathsf{Tree}\,E\,A \times \mathsf{Tree}\,E\,A)) \end{aligned}$$

(Notice that $\mathsf{Zipper}\,E$ is analogous to the type constructor LS, which is the zipper datatype for streams.) In Haskell, we can define:

```
data Path e a = Nil | Path e a :> Turn a (Tree e a)
type Turn x y = Either (x, y) (x, y)

data Zipper e a = Path e a :=| Tree e a
```

(:> is a snoc-like constructor for path structures. :=| is the pairing of a path structure and a tree into a zipper.)

The zipper datatype supports movements both up and sideways as well as down in a tree (*redoing* and *undoing* the zipper). The following upward focus shift operation in Haskell returns the zippers corresponding to the parent and the right or left sibling of the local node (unless the local node is the root of the global tree). (So we have put the parent and sibling functions into one partial function, as both are defined exactly when the local node is not the global root. This will be convenient for us.)

```
goParSibl :: Zipper e a
                  -> Maybe (Turn (Zipper e a) (Zipper e a))

goParSibl (Nil :=| as) = Nothing
goParSibl (az :> Left  (a, asR) :=| as)
          = Just (Left  (az :=| (a :< Bin as asR),
                        (az :> Right (a, as) :=| asR)))
goParSibl (az :> Right (a, asL) :=| as)
          = Just (Right (az :=| (a :< Bin asL as),
                        (az :> Left  (a, as) :=| asL)))
```

The downward focus shift operation returns the terminal, if the local node is a leaf, and the zippers corresponding to the children, if the local node is binary. (We use a Trunk structure to represent this information.)

```
goChildren :: Zipper e a -> Trunk e (Zipper e a)

goChildren (az :=| (a :< Leaf e)) = Leaf e
goChildren (az :=| (a :< Bin asL asR))
          = Bin (az :> Left  (a, asR) :=| asL)
                (az :> Right (a, asL) :=| asR)
```

This does not seem to have been mentioned in the literature, but the type constructor Zipper E is a comonad (just as LS is; in fact, the same is true of all zipper type constructors). Notably, the central operation of coextension is beautifully definable in terms of the operations goParSibl and goChildren. This is only natural, since a function taking a tree with a focus to a local value is lifted to a tree-valued function by applying it to all possible refocussings of an input tree, and that is best organized with the help of suitable operations of shifting the focus.

```
instance Comonad (Zipper e) where
  counit (_ :=| (a :< _))  = a
  cobind k d = cobindP k d :=| cobindT k d
      where cobindP k d = case goParSibl d of
              Nothing -> Nil
              Just (Left  (d', dR)) ->
                cobindP k d' :> Left  (k d', cobindT k dR)
              Just (Right (d', dL)) ->
                cobindP k d' :> Right (k d', cobindT k dL)
```

```
cobindT k d = k d :< case goChildren d of
  Leaf e -> Leaf e
  Bin dL dR -> Bin (cobindT k dL) (cobindT k dR)
```

Of course, Zipper$E$ is the comonad that structures general attribute evaluation, similarly to LS in the case of general stream-based computation.

The interpretation of coKleisli arrows as tree functions and the operation for obtaining the values of an attribute at the children are implemented essentially as for Tree$E$.

```
instance TF e Zipper where
  run k as = bs where Nil :=| bs = cobind k (Nil :=| as)

instance Synth e Zipper where
  children (_ :=| (_ :< as)) = case as of
                       Leaf e                  -> Leaf e
                       Bin (aL :< _) (aR :< _) -> Bin aL aR
```

For the children operation, we might even choose to reuse the implementation we already had for Tree$E$:

```
instance Synth e Zipper where
  children (_ :=| d) = children d
```

But differently from Tree$E$, the comonad Zipper$E$ makes it possible to also query the parent and the sibling of the current node (or to see that it is the root).

```
class Inh e d where
  parSibl :: d e a -> Maybe (Turn a a)

instance Inh e Zipper where
  parSibl (Nil    :=| _) = Nothing
  parSibl (_ :> Left  (a, aR :< _) :=| _) =
                                      Just (Left  (a, aR))
  parSibl (_ :> Right (a, aL :< _) :=| _) =
                                      Just (Right (a, aL))
```

Notice that the locality aspect of general attribute grammars (attribute values at a node are defined in terms of values of this and other attributes at neighboring nodes) is nicely supported by the local navigation operations of the zipper datatype. What is missing in the navigation operations is support for uniformity (the value of an attribute is defined in the same way everywhere in a tree). But this is provided by the coextension operation of the comonad structure on the zipper datatype. Hence, it is exactly the presence of the comonad structure that makes the zipper datatype so fit for explaining attribute evaluation.

10.3.4 Examples

We can now implement the two example attribute grammars. This amounts to rewriting the semantic equations as definitions of coKleisli arrows from the unit type.

The first grammar rewrites to the following three (mutually recursive) definitions parameterized over a comonad capable of handling purely synthesized attribute grammars (so they can be instantiated for both Tree$E$ and Zipper$E$).

```
avl :: (Comonad (d e), Synth e d) => d e () -> Bool
avl d = case children d of
             Leaf _  -> True
             Bin _ _ -> bL && bR && locavl d
               where Bin bL bR = children (cobind avl d)

locavl :: (Comonad (d e), Synth e d) => d e () -> Bool
locavl d = case children d of
             Leaf _  -> True
             Bin _ _ -> abs (hL - hR) <= 1
               where Bin hL hR = children (cobind height d)

height :: (Comonad (d e), Synth e d) => d e () -> Integer
height d = case children d of
             Leaf _  -> 0
             Bin _ _ -> max hL hR + 1
               where Bin hL hR = children (cobind height d)
```

The second grammar is rewritten completely analogously, but the definitions require a comonad that can handle also inherited attributes (so that, of our two comonads, only Zipper$E$ qualifies). Notice that the definition of the root value of the inherited attribute *numin* becomes part of the grammar description here.

```
numin  :: (Comonad (d e), Synth e d, Inh e d)
                                          => d e () -> Int
numin  d = case parSibl d of
    Nothing -> 0
    Just (Left  _) -> ni  + 1
     where Just (Left  (ni, _ )) = parSibl (cobind numin  d)
    Just (Right _) -> noL + 1
     where Just (Right (_, noL)) = parSibl (cobind numout d)

numout :: (Comonad (d e), Synth e d, Inh e d)
                                          => d e () -> Int
numout d = case children d of
    Leaf e  -> numin d
    Bin _ _ -> noR
      where Bin _ noR = children (cobind numout d)
```

We can conduct some tests, which give the desired results:

```
> let t = () :< Bin
                (() :< Bin
                       (() :< Leaf 100)
                       (() :< Bin
                              (() :< Leaf 101)
```

```
                          (() :< Leaf 102)))
              (() :< Leaf 103)

> run (\ (d :: Tree Int ()) -> (avl d, height d)) t
(False,3) :< Bin
     ((True,2) :< Bin
            ((True,0) :< Leaf 100)
            ((True,1) :< Bin
                   ((True,0) :< Leaf 101)
                   ((True,0) :< Leaf 102)))
     ((True,0) :< Leaf 103)

> run (\ (d :: Zipper Int ()) -> (numin d, numout d)) t
(0,6) :< Bin
     ((1,5) :< Bin
            ((2,2) :< Leaf 100)
            ((3,5) :< Bin
                   ((4,4) :< Leaf 101)
                   ((5,5) :< Leaf 102)))
     ((6,6) :< Leaf 103)
```

We observe that the definitions of the coKleisli arrows match the semantic equations most directly. That is, a simple attribute evaluator is obtained just by putting together a tiny comonadic core and a straightforward rewrite of the semantic equations. It is obvious that the rewriting is systematic and hence one could easily write a generic comonadic attribute evaluator for attribute grammars on our fixed context-free grammar. We refrain from doing this here.

10.3.5 Discussion

We now proceed to a short discussion of our proposal.

1. Our approach to attribute evaluation is very denotational by its spirit and our code works thanks to Haskell's laziness. There is no need for static planning of the computations based on some analysis of the grammar, attribute values are computed on demand. In particular, there is no need for static circularity checking, the price being, of course, that the evaluator will loop when a computation is circular.

But this denotational-semantic simplicity has severe consequences on efficiency. Unless some specific infrastructure is introduced to cache already computed function applications, we get evaluators that evaluate the same attribute occurrence over and over. It is very obvious from our example of AVL-hood: evaluation of *locavl* at some given node in a tree takes evaluation of *height* at all nodes below it. If we need to evaluate *locavl* everywhere in the tree, we should evaluate *height* everywhere below the root just once, but our evaluator will compute the height of each node anew each time it needs it. The very same problem is present also in the comonadic approach to dataflow computation. Simple comonadic code illustrates the meaning of dataflow computation very well, but

to achieve efficiency, one has to put in more care. Luckily, there are methods for doing so, ranging from memoization infrastructures to total reorganization of the evaluator based on tupling and functionalization transformations. We refrain from discussing these methods in the present paper.

2. Instead of relying on general recursion available in Haskell, we could forbid circularity on the syntactic level (essentially saying that an attribute value at a node cannot be defined via itself). This is standard practice in attribute grammar processors, but for us it means we can confine ourselves to using structured recursion schemes only. For purely synthesized attribute grammars, where attribute evaluation reduces to upward accumulations, we presented a solution based on structured recursion in our SFP '01 paper [UV02]. Obviously here is an analogy to syntactic circularity prevention in dataflow languages, which is also standard practice.

3. In the examples, we used incomplete pattern matches (in the where-clauses). These are guaranteed to never give a run-time error, because the coextension operation and the operations children and parSibl remember if a focal node is leaf or parent, root, left child or right child. But the type system is unaware of this. This aesthetic problem can be remedied with the generalized algebraic datatypes (GADTs) of GHC [PJWW04] (in combination with rank-2 polymorphism). For example, trees and trunks can be classified into leaves and parents at the type level by defining

```
data Tree ty e a = a :<  Trunk ty e (UTree e a)
data UTree e a = forall ty . Pack (Tree ty e a)

data Leaf
data Bin

data Trunk ty e x where
   Leaf :: e -> Trunk Leaf e x
   Bin  :: x -> x -> Trunk Bin e x
```

An analogous refinement is possible for the path structure datatype. Under this typing discipline, our pattern matches are complete.

These finer datatypes do however not solve another aesthetic problem. When trees and trunks have been made leaves or parents at the type level, it feels unnatural to test this at the level of values, as is done in the case-constructs of our code. One would instead like a typecase construct. This situation arises because our types Leaf and Bin should really be values from a doubleton type, but in Haskell value-indexed types have to be faked by type-indexed types. A real solution would be to switch to a dependently typed language and to use inductive families.

4. We finish by remarking that GADTs or inductive families are also needed to deal with multiple nonterminals in a generic attribute grammar processor capable of handling any underlying context-free grammar. For a fixed context-free grammar, mutually recursive Haskell datatypes are enough (one attributed syntax tree datatype for every nonterminal). But in the case where the underlying context-free grammar becomes a parameter, these syntax tree datatypes must be indexed

by the corresponding nonterminals, whereby each datatype in the indexed family has different constructors. In this situation, GADTs become inevitable.

10.4 RELATED WORK

The uses of coKleisli categories of comonads to describe notions of computation have been relatively few. The idea has been put forward several times, e.g., by Brookes and Geva [BG92] and by Kieburtz [Kie99], but never caught on because of a lack of compelling examples. The example of dataflow computation seems to appear first in our papers [UV05a, UV05b].

The Freyd categories / arrow types of Power and Robinson [PR97] and Hughes [Hug00] have been considerably more popular, see, e.g., [Pat03, Hug05] for overviews. The main application is reactive functional programming.

From the denotational point of view, attribute grammars have usually been analyzed proceeding from the initial algebra structure of tree types. The central observation is that an attribute evaluator is ultimately a fold (if the grammar is purely synthesized) or an application of a higher-order fold (if it also has inherited attributes) [CM79, May81]; this definition of attribute evaluation is straightforwardly implemented in a lazy functional language [Joh87, KS86]. Gibbons [Gib93, Gib00] has specifically analyzed upward and downward accumulations on trees.

Finer functional attribute grammar processors depart from the denotational approach; an in-depth analysis of the different approaches to functional attribute grammar evaluation appears in Saraiva's thesis [Sar99]. Some realistic functional attribute grammar processors are Lrc [KS98] and UUAG [BSL03].

One of the salient features of our approach is the per-attribute organization of the evaluators delivered. This is not typical to functional attribute grammar evaluators. But decomposability by attributes has been identified as desirable in the works on "aspect-oriented" attribute grammar processors by de Moor, Peyton-Jones and Van Wyk [dMPJvW99] and de Moor, K. Backhouse and Swierstra [dMBS00]. These are clearly related to our proposal, but the exact relationship is not clear at this stage. We conjecture that use of the comonad abstraction is orthogonal to the techniques used in these papers, so they might even combine.

The zipper representation of trees with a distinguished position is a piece of folklore that was first documented by Huet [Hue97]. Also related are container type constructors that have been studied by McBride and his colleagues [McB00, AAMG05].

The relation between upward accumulations and the comonad structure on trees was described by in our SFP '01 paper [UV02]. In that paper, we also discussed a basic theorem about compositions of recursively specified upward accumulations. We are not aware of any work relating attribute evaluation to comonads or arrow types.

10.5 CONCLUSIONS AND FUTURE WORK

We have shown that attribute evaluation bears a great deal of similarity to dataflow computation in that computation happens on a fixed datastructure and that the result values are defined uniformly throughout the structure with the help of a few local navigation operations to access the contexts of the argument values. As a consequence, our previous results on comonadic dataflow computation and comonadic processing of dataflow languages are naturally transported to attribute evaluation. We are very pleased about how well comonads explicate the fundamental locality and uniformity characteristics of attribute definitions that initial algebras fail to highlight. In the case of the zipper datatype, we have seen that the only thing needed to make it directly useable in attribute evaluation is to derive an explicit coextension operation from the focus shift operations.

In order to properly validate the viability of our approach, we plan to develop a proof-of-concept comonadic processor of attribute grammars capable of interpreting attribute extensions of arbitrary context-free grammars. The goal is to obtain a concise generic reference specification of attribute evaluation. We predict that the limitations of Haskell's type system may force a solution that is not as beautiful than it should ideally be, but GADTs will provide some help.

We also plan to look into systematic ways for transforming the comonadic specifications into efficient implementations (cf. the problem of efficient compilation of dataflow languages). For purely synthesized attribute grammars, a relatively straightforward generic tupling transformation should solve the problem, but the general case will be a challenge.

Acknowledgments We are very grateful to our anonymous referees for their constructive criticism and suggestions, especially in regards to related work and certain emphases of the paper.

The authors were partially supported by the Estonian Science Foundation under grant No. 5567.

REFERENCES

[AAMG05] M. Abbott, T. Altenkirch, C. McBride and N. Ghani. δ for data: differentiating data structures. *Fund. Inform.*, 65(1–2):1–28, 2005.

[AW85] E. A. Ashcroft, W. W. Wadge. *LUCID, The Dataflow Programming Language*. Academic Press, 1985.

[BG92] S. Brookes, S. Geva. Computational comonads and intensional semantics. In M. P. Fourman, P. T. Johnstone, and A. M. Pitts, eds., *Applications of Categories in Computer Science*, v. 177 of *London Math. Society Lect. Note Series*, 1–44. Cambridge Univ. Press, 1992.

[BSL03] A. Baars, S. D. Swierstra, A. Löh. UU AG system user manual. 2003.

[CM79] L. M. Chirica, D. F. Martin. An order-algebraic definition of Knuthian semantics. *Math. Syst. Theory*, 13:1–27, 1979.

[CP96] P. Caspi, M. Pouzet. Synchronous Kahn networks. In *Proc. of 1st ACM SIGPLAN Int. Conf. on Functional Programming, ICFP '96*, 226–238. ACM Press, 1996. Also in *SIGPLAN Notices*, 31(6):226–238, 1996.

[dMBS00] O. de Moor, K. Backhouse, S. D. Swierstra. First-class attribute grammars. *Informatica (Slovenia)*, 24(3):329–341, 2000.

[dMPJvW99] O. de Moor, S. Peyton Jones, E. Van Wyk. Aspect-oriented compilers. In K. Czarnecki, U. W. Eisenecker, eds., *Generative and Component-Based Software Engineering, GCSE '99 (Erfurt, Sept. 1999)*, vol. 1799 of *Lect. Notes in Comput. Sci.*, 121–133. Springer-Verlag, 1999.

[Gib93] J. Gibbons. Upwards and downwards accumulations on trees. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, eds., *Proc. of 2nd Int. Conf. on Math. of Program Construction, MPC '92 (Oxford, June/July 1992)*, vol. 669 of *Lect. Notes in Comput. Sci.*, 122–138. Springer-Verlag, 1993.

[Gib00] J. Gibbons. Generic downward accumulation. *Sci. of Comput. Program.*, 37(1–3):37–65, 2000.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud. The synchronous data flow programming language LUSTRE. *Proc. of the IEEE*, 79(9):1305–1320, 1991.

[Hue97] G. Huet. The zipper. *J. of Funct. Program.*, 7(5):549–554, 1997.

[Hug00] J. Hughes. Generalising monads to arrows. *Sci. of Comput. Program.*, 37(1–3):67–111, 2000.

[Hug05] J. Hughes. Programming with arrows. In V. Vene, T. Uustalu, eds., *Revised Lectures from 5th Int. School on Advanced Functional Programming, AFP 2004*, vol. 3622 of *Lect. Notes in Comput. Sci.*, 73–129. Springer-Verlag, 2005.

[Joh87] T. Johnsson. Attribute grammars as a functional programming paradigm. In G. Kahn, ed., *Proc. of 3rd Int. Conf. on Functional Programming and Computer Architecture, FPCA '87 (Portland, OR, Sept. 1987)*, vol. 274 of *Lect. Notes in Comput. Sci.*, 154–173. Springer-Verlag, 1987.

[Kie99] R. Kieburtz. Codata and comonads in Haskell. Unpublished draft, 1999.

[Knu68] D. Knuth. Semantics of context-free languages. *Math. Syst. Theory*, 2(2):127–145, 1968. Corrigendum, ibid., 51(1):95–96, 1971.

[KS86] M. F. Kuiper, S. D. Swierstra. Using attributed grammars to derive efficient functional programs. Techn. report RUU-CS-86-16, Dept. of Comput. Sci., Univ. f Utrecht, 1986.

[KS98] M. F. Kuiper, J. Saraiva. Lrc: a generator for incremental language-oriented tools. In K. Koskimies, ed., *Proc. of 7th Int. Conf. on Compiler Construction, CC '98 (Lisbon, March/Apr. 1998)*, vol. 1383 of *Lect. Notes in Comput. Sci.*, pp. 298–301. Springer-Verlag, 1998.

[May81] B. Mayoh. Attribute grammars and mathematical semantics. *SIAM J. of Comput.*, 10(3):503–518, 1981.

[McB00] C. McBride. The derivative of a regular type is the type of its one-hole contexts. Manuscript, 2000.

[Mog91] E. Moggi. Notions of computation and monads. *Inform. and Comput.*, 93(1):55–92, 1991.

[Pat03] R. Paterson. Arrows and computation. In J. Gibbons, O. de Moor, eds., *The Fun of Programming*, *Cornerstones of Computing*, 201–222. Palgrave Macmillan, 2003.

[PJWW04] S. Peyton Jones, G. Washburn, S. Weirich. Wobbly types: practical type inference for generalized algebraic datatypes. Technical report MS-CIS-05-26, Comput. and Inform. Sci. Dept., Univ. of Pennsylvania, 2004.

[PR97] J. Power, E. Robinson. Premonoidal categories and notions of computation. *Math. Structures in Comput. Sci.*, 7(5):453–468, 1997.

[Sar99] J. Saraiva. Purely Functional Implementation of Attribute Grammars. PhD thesis, Dept. of Comput. Sci., Utrecht University, 1999.

[UV02] T. Uustalu, V. Vene. The dual of substitution is redecoration. In K. Hammond, S. Curtis, eds., *Trends in Functional Programming 3*, 99–110. Intellect, 2002.

[UV05a] T. Uustalu, V. Vene. Signals and comonads. *J. of Univ. Comput. Sci.*, 11(7):1310–1326, 2005.

[UV05b] T. Uustalu, V. Vene. The essence of dataflow programming. In K. Yi, ed., *Proc. of 3rd Asian Symp. on Programming Languages and Systems, APLAS 2005 (Tsukuba, Nov. 2005)*, vol. 3780 of *Lect. Notes in Comput. Sci.*, 2–18. Springer-Verlag, 2005.

[Wad92] P. Wadler. The essence of functional programming. In *Conf. Record of 19th Ann. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL '92 (Albuquerque, NM, Jan. 1992)*, 1–12. ACM Press, 1992.