Comonadic functional attribute evaluation

Tarmo Uustalu[1] and Varmo Vene[2]

[1] Inst. of Cybernetics at Tallinn Univ. of Technology,
Akadeemia tee 21, EE-12618 Tallinn, Estonia; tarmo@cs.ioc.ee
[2] Dept. of Computer Science, Univ. of Tartu,
J. Liivi 2, EE-50409 Tartu, Estonia; varmo@cs.ut.ee

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.

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.

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 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 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. We are not aware of earlier comonadic or arrow-based accounts of attribute evaluation.

The paper is organized as follows. In Sec. 2, we overview our comonadic approach to dataflow computation and processing of dataflow languages. In Sec. 3, we demonstrate that the attribute evaluation paradigm can also be analyzed comonadically. Sec. 4 is a very brief review of the related work whereas Sec. 5 summarizes. Most of the development is a carried out in the programming language Haskell, directly demonstrating that the approach is implementable.

Disclaimer: This version of the paper is an extended abstract and only describes the most central constructions of our work, leaving out quite a few explanations and examples as well as large portions of the more technical material (including comonadic processing of attribute grammars).

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 $D : |\mathcal{C}| \to |\mathcal{C}|$ 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
```

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.

```
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
```

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.

3 COMONADIC ATTRIBUTE EVALUATION

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

An attribute grammar 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 (one would like to use inductive families or at least the generalized algebraic datatypes of GHC [PJWW04]), we proceed from a fixed context-free grammar with a single non-terminal S with two associated production rules

$$\begin{aligned} S &\longrightarrow E \\ S &\longrightarrow SS \end{aligned}$$

where E is a pseudo-non-terminal standing for some set of terminals. The type of attributed grammatical S-trees is $\mathsf{Tree}\,E\,A = \mu X.A \times (E + X \times X) \cong A \times \mathsf{Subs}\,E\,A$ where A is the type of the S-attributes (aggregated into records) and $\mathsf{Subs}\,E\,A = E + \mathsf{Tree}\,E\,A \times \mathsf{Tree}\,E\,A$. In Haskell, we can define:

```
data Tree e a = a :< Subs e a

type Subs e a = Trunk e (Tree e a)

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

An attribute grammar extends the underlying context-free grammar with attributes and semantic equations. These are attached to the non-terminals and the production rules of the context-free grammar. Attribute grammars are classified into purely synthesized attribute grammars and general attribute grammars (where there are also inherited attributes). 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 Tree, the idea being that the second component of a value in $\mathsf{Tree}\,E\,A \cong A \times \mathsf{Subs}\,E\,A$ can record the course of an attribute below a current node. 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)

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

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 synthesized attribute values) can be done via an operation similar to 'next'.

```
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
```

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, of course, for our tree type constructor. The course of an attribute above and around a given node lives in the type $\mathsf{Cxt}\,E\,A = \mu X.\,1 + X \times (A \times \mathsf{Tree}\,E\,A + A \times \mathsf{Tree}\,E\,A)$ of 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 records both the local value and lower and upper-and-surrounding courses of an attribute: we define $\mathsf{CxtTree}\,E\,A = \mathsf{Cxt}\,E\,A \times \mathsf{Tree}\,E\,A \cong \mathsf{Cxt}\,E\,A \times (A \times \mathsf{Subs}\,E\,A)$. (Notice that this is analogous to the type constructor LS, which is the zipper type for streams.)

```
data Cxt e a = Nil | Cxt e a :> Either (a, Tree e a)
                                       (a, Tree e a)
data CxtTree e a = Cxt e a :=| Tree e a
```

This does not seem to have been mentioned in the literature, but the type constructor $\mathsf{CxtTree}\,E$ is a comonad (just as LS is; in fact, the same is true of all zipper type constructors). And this is the comonad that structures general attribute evaluation, similarly to LS in the case of general stream-based computation.

```
instance Comonad (CxtTree e) where
  counit (_ :=| (a :< _))  = a
  cobind k d = cobindC k d :=| cobindT k d
    where cobindC k (Nil :=| _) = Nil
          cobindC k (az :> Left  (a, asR) :=| asL) =
            cobindC k d' :> Left  (k d',
                 cobindT k (az :> Right (a, asL) :=| asR))
                       where d' = az :=| (a :< Bin asL asR)
          cobindC k (az :> Right (a, asL) :=| asR) =
            cobindC k d' :> Right (k d',
                 cobindT k (az :> Left  (a, asR) :=| asL))
                       where d' = az :=| (a :< Bin asL asR)
          cobindT k d@(az :=| (a :< as)) =
            k d :< case as of
              Leaf e -> Leaf e
```

```
            Bin asL asR -> Bin
                (cobindT k (az :> Left  (a, asR) :=| asL))
                (cobindT k (az :> Right (a, asL) :=| asR))
```

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 CxtTree where
  run k as = bs where Nil :=| bs = cobind k (Nil :=| as)

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

But differently from Tree$E$, the comonad CxtTree$E$ 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 (Either (a, a) (a, a))

instance Inh e CxtTree where
  parSibl (Nil    :=| _) = Nothing
  parSibl (_ :> b :=| _) = Just $ case b of
                        Left  (a, a' :< _) -> Left  (a, a')
                        Right (a, a' :< _) -> Right (a, a')
```

Below are a few examples of attribute grammar implementations in the comonadic style: checking if a tree is AVL and preorder numbering of the nodes of a tree:

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

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

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

numin :: (Comonad (d e), Synth e d, Inh e d) => d e () -> Int
numin d = case parSibl (cobind (pair numin numout) d) of
            Nothing -> 0
```

```
            Just (Left ((ni, _), _))   -> ni + 1
            Just (Right (_, (_, noL))) -> noL + 1

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

The corresponding attribute grammars are

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

(the superscripts ℓ, b and subscripts L, R are a notational device to tell apart the different occurrences of the non-terminal S in the productions.)

From these examples it is not hard to work out a processor of attribute grammars on our fixed context-free grammar. We leave both this as well as a further discussion of the comonadic method and further examples for the full paper.

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.

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]. Gibbons [Gib93] has specifically analyzed upward and downward accumulations on trees.

Zippers, a representation of trees with a distinguished position, were invented 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]. We are not aware of any work relating attribute evaluation to comonads or arrow types.

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 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 characteristics of attribute definitions that initial algebras fail to highlight.

In order to properly validate the viability of our approach, we plan to develop a realistic comonadic processor of attribute grammars capable of interpreting attribute extensions of arbitrary context-free grammars.

Acknowledgments 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.

[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.

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

[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.

[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.

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

[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.

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

[PJWW04] S. Peyton Jones, G. Washburn, S. Weirich. Wobbly types: practical type inference for generalized algebraic datatypes. Manuscript, 2004.

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

[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. In M. A. Musicante, R. M. F. Lima, eds., *Proc. of 9th Brazilian Symp. on Programming Languages, SBLP'05 (Recife, PE, May 2005)*, 215–228. Univ. de Pernambuco, Recife, 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.*, Springer-Verlag, to appear.

[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.