Structured Computation on Trees

Tarmo Uustalu

joint work with Varmo Vene

Theory Days at Koke, 3–5 Feb. 2006

Summary

- Ever had to represent trees with a focus of attention and refocussing operations?
- The way: know or reinvent Huet's zipper (JFP 97).
- Idea: a tree with a focus = a path structure and a subtree
- Computations on trees: attribute evaluation, XML text manipulation.
- In attribute evaluation, the shape of the tree does not change and the same computation is done at every node (of the same type). The computation is of local nature.
- Attribute evaluation: comonadic computation on the comonad of the zipper, similarly to dataflow computation which is also comonadic.

Outline

- Huet's zipper
- Comonads
- Attribute evaluation as comonadic computation

Attributed binary trees

- We are interested in attributed binary trees. Mathematically,

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

- In Haskell,

```
data Tree  e a = a :< Trunk e (Tree e a)
data Trunk e x = Leaf e | Bin x x
```

The zipper

- A path structure determines everything above and around a node in a hypothetical tree.

  It is a path (list of turns) decorated with node attributes and side trees.

  The list is a snoc-list: we want to read it left-to-right (= down from root to focus), but analyse it right-to-left (= up from focus to root).

- Mathematically,

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

- In Haskell,

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

data Either z w = Left z | Right w
```

- A tree with a focus = a path structure and a (sub)tree:

$$\mathsf{CxtTree}\, E\, A = \mathsf{Cxt}\, E\, A \times \mathsf{Tree}\, E\, A$$

  or,

```
data CxtTree e a = Cxt e a :=| Tree e a
```

Zipper navigation: doing and undoing the zipper

- Moving up (doing):

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

- Moving down (undoing):

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

Comonads

- Comonads are a formal dual of monads. Monads are extensively used in functional programming to structure effectful computation.
- A *comonad* is a a type constructor D with two polymorphic functions $\epsilon : D\,A \to A$ (the *counit*) and $-^{\dagger} : (D\,A \to B) \to (D\,A \to D\,B)$ (the *coKleisli extension*), satisfying certain laws.
- In Haskell,

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

- A function $f : D\,A \to B$ is called a *coKleisli arrow* (of D) from $A \to B$ (notation $f : A \rightsquigarrow B$).
- The counit is the identity coKleisli arrow, the coKleisli extension gives that they can be composed. The comonad laws guarantee that composition is associative and has the identity of as the left and right unit.
- Any function $f : A \to B$ can be lifted to a coKleisli arrow $Jf : A \rightsquigarrow B$: take $Jf = f \circ \epsilon_A$

- The Kleisli arrows of a comonad correspond to context-dependent functions.
- A context-dependent function from A to B is a function $DA \to B$ i.e., a coKleisli map $A \rightsquigarrow B$.
- DA is the type of contextually situated values of type A.
- The counit $\epsilon_A : DA \to A$ discards the context of its input.
- The coextension $k^\dagger : DA \to DB$ of a function $k : DA \to B$ duplicates the context of its input (to feed it to k and still have a copy left).
- Examples: product comonad, comonads of dataflow computation (Uustalu, Vene, APLAS 2005)
- Examples of today: trees and zippers and attribute evaluation

Attribute grammars

- An attribute grammar is a CF grammar augmented with attributes and semantic equations.
- We consider a fixed CF grammar with one non-terminal S and production rules

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

  E is a pseudoterminal.
- For an attribute type A for S-nodes, the type of attributed S-trees is $\mathsf{Tree}\, E\, A$.

Purely synthesized attribute grammars

- For a purely synthesized attribute grammar, the local value of a defined attribute of a semantic equation can explicitly depend on the local and children-node values of the defining attributes.
- The relevant comonad is $\mathsf{Tree}\,E$, where the value under focus is the root atribute value and those in the subtrees form are its context.
- In Haskell,

```
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 Kleisli arrows represent tree functions:

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

instance TF e Tree where
  run = cobind
```

- Looking up the values at the children:

```
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
```

General attribute grammars

- Here we need a more permissive notion of context. This is provided by the zipper.
- Now CxtTree E is a comonad as well!

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

- CoKleisli arrows can be interpreted as tree functions and there is an operation for getting the values of the attribute at the children of a node.

```
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
```

- There is also an operation that checks whether the local node is the root or not, and if it is not, obtains the value of the attribute at the parent and the sibling.

```
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')
```

Examples

- Checking a tree for AVLness.
- Attribute grammar presentation:

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

- Comonadic Haskell:

```
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)
```

- Preorder numbering of the nodes of a tree.
- Attribute grammar presentation:

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

- Comonadic Haskell:

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

Conclusions

- The zipper datatype hides a comonad. This is exactly the comonad one needs to structure attribute evaluation.
- GADTs+typecase or dependent types needed to smoothly deal with multiple non-terminals, multiple rules for individual non-terminals.
- To do: Comonadically interpreted AG definition language.