Precise comonads for dataflow computation and tree transformations

*Not obsessed with monads so much more, still obsessed with comonads* . . .

Tarmo Uustalu, Tallinn
Joint work with Varmo Vene, Tartu

EffTT, Tallinn, 13–14 December 2007
Dataflow computation

- Dataflow computations = discrete-time signal transformations = stream functions.
- The output value at a time instant (stream position) is determined by the input value at the same instant (position) plus further input values.
- General dataflow, dependence on past and future / causal dataflow, dependence on past alone.
- Lucid, French synchronous languages (Lustre, Lucid Synchrone).
- (Related to Mealy machines.)
Example dataflow programs

\[
\begin{align*}
pos &= 0 \text{ fby } (\text{pos } + 1) \\
\text{sum } x &= x + (0 \text{ fby } (\text{sum } x)) \\
\text{fact} &= 1 \text{ fby } (\text{fact } \ast (\text{pos } + 1)) \\
\text{fibo} &= 0 \text{ fby } (\text{fibo } + (1 \text{ fby } \text{fibo}))
\end{align*}
\]

<table>
<thead>
<tr>
<th></th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>...</td>
</tr>
<tr>
<td>17 fby 5</td>
<td>17</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>...</td>
</tr>
<tr>
<td>pos</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>...</td>
</tr>
<tr>
<td>sum pos</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>15</td>
<td>21</td>
<td>28</td>
<td>...</td>
</tr>
<tr>
<td>fact</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>24</td>
<td>120</td>
<td>720</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>fibo</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>...</td>
</tr>
</tbody>
</table>

('fby' (‘followed by’) means unit delay)
Tree transformations

- Attribute evaluation = tree relabelling transformations.
- The label at a position in the output tree is determined by the label in the same position in the input tree plus further labels of the input tree (below or anywhere).
- Purely synthesized, dependence on nodes below / general attribute grammars, dependence on nodes both below and above-aside.
- (Related to relabelling tree transducers.)
Example attribute grammar

\[
S^\ell \rightarrow E \\
S^b \rightarrow S^b_L S^b_R
\]

\[
S^\ell .avl = \text{tt} \\
S^b .avl = S^b_L .avl \land S^b_R .avl \land S^b .locavl \\
S^\ell .locavl = \text{tt} \\
S^b .locavl = |S^b_L .height - S^b_R .height| \leq 1 \\
S^\ell .height = 0 \\
S^b .height = \max(S^b_L .height, S^b_R .height) + 1
\]
Context-dependent computation

- Common to both dataflow computation and tree transformations is computation in a datastructure (container).
- The shape of the datastructure is kept, the computation for every position is
  - local, although context-dependent,
  - uniform, follows the same rule.
This talk

- Moggi’s, late 1980s: analysis of different notions of effectful (cbv) computation in terms Kleisli categories of strong monads.

- Brookes, Geva and Stone, early 1990s: coKleisli categories of “computational” comonads for “intensional semantics”.

- Our 2 cent: CoKleisli categories of symmetric (semi)monoidal comonads are a setting to analyse notions of context-dependent computation such as dataflow computation and tree transformations . . .

- . . . and sometimes you may want a comonad on a functor category instead of your base category for things to work as they should.
Outline

- Comonads and context-dependent computation (cf monads and effectful computation)
- Symmetric (semi)monoidal monads and context-dependent computation with products and function spaces (cf strong monads and effectful computation with products and function spaces)
- Semantics of context-dependent languages (cf Kleisli semantics of effectful languages)
- Refined comonads on functor categories
- Examples: dataflow computation, attribute evaluation
Comonads

- Comonads are the dual of monads.
- A comonad is
  - a functor $D : C \to C$ (the underlying functor),
  - a natural transformation $\varepsilon : D \to \text{Id}_C$ (the counit),
  - a natural transformation $\delta : D \to DD$ (the comultiplication)

satisfying these conditions:

- In other words, a comonad is a comonoid in $[C, C]$ (a monoid in $[C, C]^{\text{op}}$).
CoKleisli category of a comonad

A comonad $D$ on a category $C$ induces a category $\textbf{CoKl}(D)$ called the coKleisli category of $D$ defined by

- an object is an object of $C$,
- a map of from $A$ to $B$ is a map of $C$ from $DA$ to $B$,
- $\text{id}^D_A = \text{df} \ DA \xrightarrow{\varepsilon_A} A$,
- if $k : A \to^D B$, $\ell : B \to^D C$, then
  \[ \ell \circ^D k = \text{df} \ DA \xrightarrow{k^\dagger} DB \xrightarrow{\ell} C \]
  where
  \[ k^\dagger = \text{df} \ DA \xrightarrow{\mu_A} DDA \xrightarrow{Dk} DB. \]

- From $C$ there is an identity-on-objects inclusion functor $J$ to $\textbf{CoKl}(D)$, defined on maps by
  - if $f : A \to B$, then
    \[ Jf = \text{df} \ DA \xrightarrow{\varepsilon_A} A \xrightarrow{f} B = DA \xrightarrow{Df} DB \xrightarrow{\varepsilon_B} B. \]
  - The functor $J$ has a left adjoint $U : \textbf{CoKl}(D) \to C$ given by $UA = \text{df} \ DA$, if $k : A \to^D B$, then
    \[ Uk = \text{df} \ DA \xrightarrow{k^\dagger} DB. \]
Comonadic notions of computation

- We think of $C$ as the category of pure functions and of $DA$ as the type of coeffectful computations of values of type $A$ (values in context).
- $\text{CoKl}(D)$ is the category of context-dependent functions.
- $\varepsilon_A : DA \rightarrow A$ is the identity on $A$ seen as trivially context-dependent (discarding the context).
- $Jf : DA \rightarrow B$ is a general pure function $f : A \rightarrow B$ regarded as trivially context-dependent.
- $\delta_A : DA \rightarrow DDA$ blows the context of a value up (duplicates the context).
- $k^\dagger : DA \rightarrow DB$ is a context-dependent function $k : DA \rightarrow B$ extended into one that can output a value in a context (e.g., for a postcomposed context-dependent function).
Simplest (computational) examples

- Product comonad, for dependency on an environment:
  - \( DA =_{\text{df}} A \times E \) where \( E \) is an object of \( \mathcal{C} \),
  - \( \varepsilon_A =_{\text{df}} A \times E \xrightarrow{\text{fst}} A \),
  - \( \delta_A =_{\text{df}} A \times E \xrightarrow{\langle \text{id}, \text{snd} \rangle} (A \times E) \times E \).

This is the dual of the coproduct monad for exceptions.

- It is not very interesting, as \( \text{CoKl}(D) \cong \text{Kl}(T) \) for \( TA =_{\text{df}} E \Rightarrow A \).
Exponent comonad:

- \( DA =_{df} E \Rightarrow A \) where \((E, e, m)\) is a monoid in \( C \),
- \( \varepsilon_A =_{df} (E \Rightarrow A)^{ur^{-1}} \xrightarrow{id \times e} (E \Rightarrow A) \times 1 \xrightarrow{id \times m} (E \Rightarrow A) \times E \xrightarrow{ev} A, \)
- \( \delta_A =_{df} \Lambda((E \Rightarrow A) \times E) \xrightarrow{a} (E \Rightarrow A) \times (E \times E) \xrightarrow{id \times m} (E \Rightarrow A) \times E \xrightarrow{ev} A) \),

Interesting special cases are \((E, e, m) =_{df} (Nat, 0, +)\) and \((E, e, m) =_{df} (Nat, 0, \text{max})\).
Costate comonad:

- \( DA =_{df} (P \Rightarrow A) \times P \) where \( P \) is an object of \( C \),
- \( \varepsilon_A =_{df} (P \Rightarrow A) \times P \stackrel{ev}{\rightarrow} A \),
- \( \delta_A =_{df} (P \Rightarrow A) \times P \stackrel{\text{coev} \times \text{id}}{\rightarrow} (P \Rightarrow ((P \Rightarrow A) \times P)) \times P \).

This comonad arises from the adjunction \( P \times - \dashv P \Rightarrow - \). Composition the other way around gives the state monad \( TA =_{df} P \Rightarrow (A \times P) \).
Comonads for dataflow computation

We are interested in general/causal/anticausal stream functions \( \text{Str}A \to \text{Str}B \) where

\[
\text{Str}A = \text{df} \nu X.A \times X
\]

which we would like to see as context-dependent functions from \( A \) to \( B \).

Streams are naturally isomorphic to functions from natural numbers:

\[
\text{Str}A = \text{df} \nu X.A \times X \cong \text{Nat} \Rightarrow A
\]

\text{General} stream functions \( \text{Str}A \to \text{Str}B \) are thus in natural bijection with maps \((\text{Nat} \Rightarrow A) \times \text{Nat} \to B\).
The functor

\[ DA =_{df} (Nat \Rightarrow A) \times Nat \]

is a comonad (streams with a position comonad), a special case of the costate comonad, so maps \((Nat \Rightarrow A) \times Nat \to B\) are coKleisli maps.

The coKleisli identities and composition agree with the stream function identities and composition.

Important operations supported are fby : \(A \times DA \to A\) and next : \(DA \to A\) for unit delay and anticipation.
Comonad for general dataflow, concretely:

\[ DA =_{\text{df}} (\text{Nat} \Rightarrow A) \times \text{Nat} \]

\[ \varepsilon_A : (\text{Nat} \Rightarrow A) \times \text{Nat} \rightarrow A \]
\[ (f, n) \mapsto f \; n \]

\[ \delta_A : (\text{Nat} \Rightarrow A) \times \text{Nat} \rightarrow (\text{Nat} \Rightarrow ((\text{Nat} \Rightarrow A) \times \text{Nat})) \times \text{Nat} \]
\[ (f, n) \mapsto (\lambda m. (f, m), n) \]

\[ \text{fby}_A : A \times ((\text{Nat} \Rightarrow A) \times \text{Nat}) \rightarrow A \]
\[ (a_{00}, (f, 0)) \mapsto a_{00} \]
\[ (a_{00}, (f, n + 1)) \mapsto f \; n \]

\[ \text{next}_A : (\text{Nat} \Rightarrow A) \times \text{Nat} \rightarrow A \]
\[ (f, n) \mapsto f(n + 1) \]
A position in a stream splits it into two parts: elements before and after (and including) that position:

\[(\text{Nat} \Rightarrow A) \times \text{Nat} \cong \text{List}A \times \text{Str}A \cong \text{List}A \times (A \times \text{Str}A)\]

Accordingly, causal stream functions are coKleisli maps of the comonad

\[DA =_{df} \text{List}A \times A \cong \text{NEList}A =_{df} \mu X. A \times (1 + X)\]

(cofree recursive comonad on \(HX =_{df} 1 + X\), nonempty list comonad).

and anticausal stream functions are coKleisli maps of the comonad

\[DA =_{df} \text{Str}A \cong \text{Nat} \Rightarrow A\]

(stream comonad) which is a special case of the exponent comonad \(DA =_{df} S \Rightarrow A\) with \((S, e, m) =_{df} (\text{Nat}, 0, +)\).

The nonempty list comonad supports fby, the stream comonad supports next.
Comonad for causal dataflow, concretely:

\[ DA = \text{df} \text{ NEList } A \]

\[ \varepsilon_A : \text{ NEList } A \rightarrow A \]
\[ (a_0, \ldots, a_{n-1}, a_n) \mapsto a_n \]

\[ \delta_A : \text{ NEList } A \rightarrow \text{ NEList } (\text{ NEList } A) \]
\[ (a_0, \ldots, a_{n-1}, a_n) \mapsto ((a_0), \ldots, (a_0, \ldots, a_{n-1}), (a_0, \ldots, a_{n-1}, a_n)) \]

\[ \text{fby}_A : A \times \text{ NEList } A \rightarrow A \]
\[ (a_{00}, (a_0)) \mapsto a_{00} \]
\[ (a_{00}, (a_0, \ldots, a_n, a_{n+1})) \mapsto a_n \]
Comonad for anticausal dataflow, concretely:

\[ DA =_{df} \text{Str}A \]

\[ \varepsilon_A : \text{Str}A \rightarrow A \]
\[ (a_n, a_{n+1}, \ldots) \mapsto a_n \]

\[ \delta_A : \text{Str}A \rightarrow \text{Str} \left( \text{Str}A \right) \]
\[ (a_n, a_{n+1}, \ldots) \mapsto ((a_n, a_{n+1}, \ldots), (a_{n+1}, a_{n+2}, \ldots), \ldots) \]

\[ \text{next}_A : \text{Str}A \rightarrow A \]
\[ (a_n, a_{n+1}, \ldots) \mapsto a_{n+1} \]
Comonads for relabelling tree transformations

- Let $H : C \to C$. Define

  $$\text{Tree}A = \mu X.A \times HX$$

- We are interested in relabelling functions $\text{Tree}A \to \text{Tree}B$.
  (Alt. we can define $\text{Tree}^\infty A = \nu X.A \times HX$ and interest ourselves in relabelling functions $\text{Tree}^\infty A \to \text{Tree}^\infty B$.)

- Comonad for general relabelling functions:

  $$DA = \mu X.A \times HX$$

  where

  $$\text{Path}A = \text{List}(A \times H'(\text{Tree}A))$$

  (Huet’s zipper).

  E.g., for $HX = 1 + X \times X$, $H'X \cong 2 \times X$ and $\text{Path}A \cong \text{List}(A \times 2 \times \text{Tree}A)$.
Comonad for bottom-up relabelling functions:

\[ DA \overset{\text{df}}{=} \text{TreeA} \]

The important operations are those for navigation in a zipper.
Comonad for general relabelling of containers

- Streams and trees are a special case of containers, i.e., functors
  \[ FA =_{df} \coprod_{s \in S} (P_s \Rightarrow A) \]

- Shape-preserving functions \( FA \rightarrow FB \) are families of maps
  \[ (P_s \Rightarrow A \rightarrow P_s \Rightarrow B)_{s \in S}, \]
  i.e., maps \( \coprod_{s \in S} ((P_s \Rightarrow A) \times P_s) \rightarrow B. \)

- The functor
  \[ DA =_{df} \coprod_{s \in S} ((P_s \Rightarrow A) \times P_s) \cong F'A \times A \]
  is the comonad here.
Symmetric monoidal comonads

- A **strong/lax symmetric monoidal functor** between symmetric monoidal categories \((C, I, \otimes)\) and \((C', I', \otimes')\) is
  - a functor on \(F : C \to C'\)
  - together with an isomorphism/map \(e : I' \to FI\)
  - and a natural isomorphism/transformation with components \(m_{A,B} : FA \otimes' FB \to F(A \otimes B)\)

satisfying

\[
\begin{align*}
FA \otimes' I' & \xrightarrow{id \otimes' e'} FA \otimes' FI \xrightarrow{m_{A,I}} F(A \otimes I) & FA \otimes' FB & \xrightarrow{m_{A,B}} F(A \otimes B) \\
\downarrow ur_{FA}' & & \downarrow Fur_A & & \downarrow Fc_{A,B} \\
FA & \quad FA & FA & \quad FB \otimes' FA & \xrightarrow{m_{B,A}} F(B \otimes A) \\
\end{align*}
\]

\[
\begin{align*}
(FA \otimes' FB) \otimes' FC & \xrightarrow{m_{A,B} \otimes id} F(A \otimes B) \otimes' FC \xrightarrow{m_{A\otimes B,C}} F((A \otimes B) \otimes C) & \\
\downarrow a_{FA,FB,FC}' & & \downarrow Fa_{A,B,C} \\
FA \otimes' (FB \otimes' FC) & \xrightarrow{id \otimes m_{B,C}} FA \otimes' F(B \otimes C) & \xrightarrow{m_{A,B \otimes C}} F(A \otimes (B \otimes C))
\end{align*}
\]
A symmetric monoidal natural transformation between two (strong or lax) symmetric monoidal functors \((F, e, m), (G, e', m')\) is a natural transformation \(\tau : F \to G\) satisfying

\[
\begin{array}{ccc}
I' & \xrightarrow{e} & FI \\
\downarrow & & \downarrow \tau I \\
I' & \xrightarrow{e'} & GI \\
\end{array} \quad \begin{array}{ccc}
FA \otimes' FB & \xrightarrow{m_{A,B}} & F(A \otimes B) \\
\downarrow \tau_{A\otimes B} & & \downarrow \tau_{A\otimes B} \\
GA \otimes' GB & \xrightarrow{m'_{A,B}} & G(A \otimes B) \\
\end{array}
\]

A strong/lax symmetric monoidal comonad on a symmetric monoidal category \((C, I, \otimes)\) is a comonad \((D, \varepsilon, \delta)\) where \(D\) is a strong/lax symmetric monoidal functor (with \(I, \otimes\) preserved by \(e, m\)) and \(\varepsilon, \delta\) are symmetric monoidal natural transformations.
Examples revisited

- $DA =_{df} A \times E$ is lax symmetric monoidal as soon as $E$ carries a commutative monoid structure.
- $DA =_{df} S \Rightarrow E$ is strong symmetric monoidal.
- Hence $DA =_{df} \text{Str}A \cong \text{Nat} \Rightarrow A$ is strong symmetric monoidal too.
- $DA =_{df} \text{List}A \times A$ is only lax symmetric semimonoidal . . .

\[
e : 1 \rightarrow \text{NEList}1
\]
\[
() \mapsto ?
\]

\[
m_{A,B} : \text{NEList}A \times \text{NEList}B \rightarrow \text{NEList}(A \times B)
\]
\[
((a_0, \ldots, a_n), (b_0, \ldots, b_n)) \mapsto ((a_0, b_0), \ldots, (a_n, b_n))
\]
\[
((a_0, \ldots, a_n), (b_0, \ldots, b_m)) \mapsto \text{perhaps } ((a_n, a_m))
\]
CoKleisli categories and Cartesian closed structure

- Let $D$ be a comonad on a Cartesian closed category $C$.
- How much of the structure of $C$ does $\text{CoKl}(D)$ inherit?
- Since $J : C \to \text{CoKl}(D)$ is a right adjoint and preserves limits, $\text{CoKl}(D)$ inherits the products of $C$. Explicitly, we can define

$$
\begin{align*}
1^D &= \text{df} \ 1 \\
!^D &= \text{df} \ ! \\
A \times^D B &= \text{df} \ A \times B \\
fst^D &= \text{df} \ fst \circ \varepsilon \\
snd^D &= \text{df} \ snd \circ \varepsilon \\
\langle k_0, k_1 \rangle^D &= \text{df} \ \langle k_0, k_1 \rangle
\end{align*}
$$
If $D$ is $(1, \times)$ strong/lax symmetric semimonoidal, then we can also define

$$A \Rightarrow^D B =_{df} DA \Rightarrow B$$

$$\text{ev}^D =_{df} \text{ev} \circ \langle \varepsilon \circ D\text{fst}, D\text{snd} \rangle$$

$$\Lambda^D(k) =_{df} \Lambda(k \circ m)$$

$$D((DA \Rightarrow B) \times A) \xrightarrow{\langle \varepsilon \circ D\text{fst}, D\text{snd} \rangle} (DA \Rightarrow B) \times DA \xrightarrow{\text{ev}} B$$

$$DC \times DA \xrightarrow{m} D(C \times A) \xrightarrow{k} B$$

$$DC \xrightarrow{\Lambda(k \circ m)} DA \Rightarrow B$$
Using a strength (if available) is not a good idea: We have no multiplication

\[ DC \times DA \xrightarrow{sl} D(C \times DA) \xrightarrow{Dsr} DD(C \times A) \xrightarrow{?} D(C \times A) \]

and applying \( \varepsilon \) or \( D\varepsilon \) gives a solution where the order of arguments of a function is important and contexts do not combine:

\[ DC \times DA \xrightarrow{id \times \varepsilon} DC \times A \xrightarrow{sl} D(C \times A) \]

or

\[ DC \times DA \xrightarrow{\varepsilon \times id} C \times DA \xrightarrow{sr} D(C \times A) \]
If $D$ is strong semimonoidal (in which case it is automatically strong symmetric semimonoidal as well), then $A \Rightarrow^D -$ is right adjoint to $- \times^D A$ and hence $\Rightarrow^D$ is an exponent functor:

$$
\begin{align*}
D(C \times A) & \to B \\
DC \times DA & \to B \\
DC & \to DA \Rightarrow B
\end{align*}
$$

This is the case, e.g., if $DA \cong \nu X. A \times (E \Rightarrow X)$ for some $E$ (e.g., $DA \cong \text{Str} A \cong \nu X. A \times (1 \Rightarrow X)$).
Often however (if we do not take care), \( D \) is only lax symmetric (semi)monoidal.

Then it suffices to have \((e \text{ and) } m \) satisfying

\[
\begin{align*}
DA & \cong DA & DA & \cong DA \\
!_{DA} & \downarrow & D!_A & \Delta_{DA} & \downarrow & D\Delta_A \\
1 & \xrightarrow{e} & D1 & DA \times DA & \xrightarrow{m_{A,A}} & D(A \times A)
\end{align*}
\]

\(\xrightarrow{m_{A,B}} \langle D\text{fst}, D\text{snd} \rangle = \text{id}_{D(A \times B)}\).

Then \( \Rightarrow^D \) is a weak exponent operation on objects.
CoKleisli semantics

As in the case of Kleisli semantics, we interpret the simply typed lambda-calculus into $\text{CoKI}(D)$ in the standard way, using its Cartesian (pre)closed structure, getting

\[
\begin{align*}
\llbracket K \rrbracket^D &= \text{df} \quad \text{an object of } \text{CoKI}(D) \\
\llbracket 1 \rrbracket^D &= \text{df} \quad 1^D \\
\llbracket A \times B \rrbracket^D &= \text{df} \quad \llbracket A \rrbracket^D \times^D \llbracket B \rrbracket^D \\
&= \llbracket A \rrbracket^D \times \llbracket B \rrbracket^D \\
\llbracket A \Rightarrow B \rrbracket^D &= \text{df} \quad \llbracket A \rrbracket^D \Rightarrow^D \llbracket B \rrbracket^D \\
&= D \llbracket A \rrbracket^D \Rightarrow \llbracket B \rrbracket^D \\
\llbracket C \rrbracket^D &= \text{df} \quad \llbracket C_0 \rrbracket^D \times^D \ldots \times^D \llbracket C_{n-1} \rrbracket^D \\
&= \llbracket C_0 \rrbracket^D \times \ldots \times \llbracket C_{n-1} \rrbracket^D
\end{align*}
\]
\[
\begin{align*}
\llbracket (x) \ x_i \rrbracket^D & \ =_{df} \ \pi_i^D \\
\llbracket (x) \ let \ x \leftarrow t \ in \ u \rrbracket^D & \ =_{df} \ \llbracket (x, x) \ u \rrbracket^D \circ \langle \text{id}^D, \llbracket (x) \ t \rrbracket^D \rangle^D \\
\llbracket (x) \ (\ ) \rrbracket^D & \ =_{df} \ !^D \\
\llbracket (x) \ \text{fst}(t) \rrbracket^D & \ =_{df} \ \text{fst}^D \circ \llbracket (x) \ t \rrbracket^D \\
\llbracket (x) \ \text{snd}(t) \rrbracket^D & \ =_{df} \ \text{snd}^D \circ \llbracket (x) \ t \rrbracket^D \\
\llbracket (x) \ (t_0, t_1) \rrbracket^D & \ =_{df} \ \langle \llbracket (x) \ t_0 \rrbracket^D, \llbracket (x) \ t_1 \rrbracket^D \rangle^D \\
\llbracket (x) \ \lambda x t \rrbracket^D & \ =_{df} \ \Lambda^D(\llbracket (x, x) \ t \rrbracket^D) \\
\llbracket (x) \ t \ u \rrbracket^D & \ =_{df} \ \text{ev}^D \circ \langle \llbracket (x) \ t \rrbracket^D, \llbracket (x) \ u \rrbracket^D \rangle^D \\
\end{align*}
\]
Constructs specific to a particular notion of context are interpreted specifically.

E.g., for the constructs of a general/causal/anticausal dataflow language we can use the appropriate comonad and define:

\[
\begin{align*}
\llbracket (x) \ t_0 \ fby \ t_1 \rrbracket^D &= \text{df} \quad fby \circ \langle \llbracket (x) \ t_0 \rrbracket^D, (\llbracket (x) \ t_1 \rrbracket^D)^\dagger \rangle \\
\llbracket (x) \ next \rrbracket^D &= \text{df} \quad \text{next} \circ (\llbracket (x) \ t \rrbracket^D)^\dagger
\end{align*}
\]
Again, we have welldefinedness / soundness of typing, in the form $\_ : C \vdash t : A$ implies $[(\_ t)^D : [C]^D \to^D [A]^D]$. Moreover, all equations of the lambda-calculus are validated for a strong semimonoidal comonad, but not in the lax situation.

For a closed term $\vdash t : A$, soundness of typing says that $[t]^D : 1 \to^D [A]^D$, i.e., $D1 \to [A]^D$, so closed terms are evaluated relative a contextuated value of the unit type.

If $D$ is monoidal (not just semimonoidal), we have a canonical choice $e : 1 \to D1$.

In case of general or causal stream functions, an element of $D1$ is a list over 1, i.e., a natural number, the time elapsed.
Is this semantics right?

- Right wrt. what? We could compare the comonadic generic denotational semantics with some other generic semantics, ... if we had we one (e.g., operational).
- Or we can compare the comonadic denotational semantics of a specific language to its standard denotational semantics.
- First-order dataflow languages: The comonadic and standard (stream-function) semantics agree fully.
- Higher-orderness: How to combine dataflow constructs and higher-orderness has been unclear. We get a neat semantics of the “natural” higher-order extension of the first-order languages from mathematical considerations (cf. Colaço, Pouzet’s design with two flavors of function spaces).
Issues

- Inaccuracy: Dataflow computation and tree transformations can be analyzed in terms of strong monoidal comonads on \([\mathcal{I}, \mathcal{C}]\) with \(\mathcal{I}\) some small category.
- Coproducts and recursion: General recursion vs. guarded recursion for cofree recursive comonads.
- “Dual” Lawvere theories and arrow types/Freyd categories: preclosed rather than premonoidal structure is of interest (with closedness a la Eilenberg-Kelly).
- Comonad resolutions other than coKleisli.
- Operational semantics.
- Combining effects and context-dependence: distributive laws and biKleisli categories, e.g., for clocked dataflow.
The inaccuracy problem

- That the comonads for causal and general dataflow are not strong symmetric monoidal and the coKleisli categories not cartesian closed is . . . (perhaps) wrong: the function space is “too large” for poor reasons.

- We haven’t exploited that stream functions don’t change the shape of a given element (contextually situated value) of $DA =_{df} \text{List}A \times A$ or $DA =_{df} \text{List}A \times \text{Str}A$.

- For taking advantage of this, a comonad on a functor category can be used.
A “precise” comonad for causal dataflow

Instead of a comonad on a base category $\mathcal{C}$, define one on $[\omega, \mathcal{C}]$ where $\omega$ is the poset of natural numbers.

$$(DA)_n = \text{df} \prod_{j=0}^{n} A_j$$

$$(\varepsilon_A)_n : (DA)_n \rightarrow A_n$$

$$(a_0, \ldots, a_n) \mapsto a_n$$

$$(\delta_A)_n : (DA)_n \rightarrow \prod_{j=0}^{n} (DA)_j$$

$$(a_0, \ldots, a_n) \mapsto ((a_0), \ldots, (a_0, \ldots, a_n))$$

$$(fby_A)_n : A_0 \times (DA)_n \rightarrow A_n$$

$$(a_{00}, (a_0)) \mapsto a_{00}$$

$$(a_{00}, (a_0, \ldots, a_n, a_{n+1})) \mapsto A_{n\rightarrow n+1} a_n$$
This comonad is unproblematically strong symmetric monoidal in the right way:

\[
e_n : 1 \rightarrow (D1)_n
\]

\[
() \mapsto ((), \ldots, ())
\]

\[
\text{n+1 times}
\]

\[
(m_{A,B})_n : (DA)_n \times (DA)_n \rightarrow (DA)_n
\]

\[
((a_0, \ldots, a_n), (a'_0, \ldots, a'_n)) \mapsto ((a_0, a'_0), \ldots, (a_n, a'_n))
\]

Of course there is more to worry about, e.g., \(DA\) must be functorial, \(\varepsilon_A, \delta_A\) etc. must be natural for any functor \(A\).

\[
(DA)_{n \rightarrow n+1} : (DA)_n \rightarrow (DA)_{n+1}
\]

\[
(a_0, \ldots, a_n) \mapsto (a_0, A_{0 \rightarrow 1}a_0, \ldots, A_{n \rightarrow n+1}a_n)
\]
Related: Semantics of intuitionistic linear and modal logic

️ Strong symmetric monoidal comonads (and strong monads) are central in the semantics of intuitionistic linear logic and modal logic to interpret the ! and □ (◇) operators.

️ Linear logic: Benton, Bierman, de Paiva, Hyland; Bierman; Benton; Mellies; Maneggia; etc.

️ Modal logic: Bierman, de Paiva.

️ Applications to staged computation and semantics of names: Pfenning, Davies, Nanevski.
Conclusions

- Not just “intensional semantics”, but also several important and classical “context-dependent” notions of computation can be analyzed systematically in terms of comonads.
- For corresponding extensions of the lambda calculus, a “dual” Moggi-style semantics applies.
- Systematic approach, generic analysis of different notions of computation.
- Known language designs/semantics for these notions quality-checked against category-theoretic criteria of canonicity.
- New designs/semantics, e.g., categorically-motivated semantics of higher-orderness for dataflow languages, neater than the earlier proposals.