Layer by Layer: combining monads

Fredrik Dahlqvist, Alexandra Silva, Louis Parlant

October 16, 2018
Motivation: ProbNetKAT
Motivation: ProbNetKAT

A simple network:
Motivation: ProbNetKAT

A simple network:

Topology: \[ t = (sw = S_1; pt = 2; (sw \leftarrow S_2; pt \leftarrow 1) \oplus .9 \text{ drop}) \]
\[ \land (sw = S_1; pt = 3; sw \leftarrow S_3; p \leftarrow 1) \]
\[ \land (sw = S_2; pt = 4; sw \leftarrow S_4; p \leftarrow 2) \]
\[ \land (sw = S_3; pt = 4; sw \leftarrow S_4; p \leftarrow 3) \]
Motivation: ProbNetKAT

A simple network:

Topology:

\[ t = (sw = S_1; pt = 2; (sw \leftarrow S_2; pt \leftarrow 1) \oplus .9 \text{ drop}) \]
\[ \& (sw = S_1; pt = 3; sw \leftarrow S_3; p \leftarrow 1) \]
\[ \& (sw = S_2; pt = 4; sw \leftarrow S_4; p \leftarrow 2) \]
\[ \& (sw = S_3; pt = 4; sw \leftarrow S_4; p \leftarrow 3) \]

Forwarding policy: \[ p = (sw = S_1; pt \leftarrow 2)\&(sw = S_2; pt \leftarrow 4) \]
Motivation: ProbNetKAT

A simple network:

Topology:
\[ t = (sw = S_1; pt = 2; (sw \leftarrow S_2; pt \leftarrow 1) \oplus 0.9 \text{ drop}) \]
\[ \& (sw = S_1; pt = 3; sw \leftarrow S_3; p \leftarrow 1) \]
\[ \& (sw = S_2; pt = 4; sw \leftarrow S_4; p \leftarrow 2) \]
\[ \& (sw = S_3; pt = 4; sw \leftarrow S_4; p \leftarrow 3) \]

Forwarding policy:
\[ p = (sw = S_1; pt \leftarrow 2) \& (sw = S_2; pt \leftarrow 4) \]

A packet reaches \( S_4 \): \((t; p)^*; (sw = S_4)\)
Motivation: ProbNetKAT

- Syntactically, ProbNetKAT is a kind of Kleene algebra with probabilistic choice $\oplus_\lambda$
Motivation: ProbNetKAT

- Syntactically, ProbNetKAT is a kind of Kleene algebra with probabilistic choice $\oplus_\lambda$
- BUT! The denotation of the operator $\&$ is odd...
Motivation: ProbNetKAT

- Syntactically, ProbNetKAT is a kind of Kleene algebra with probabilistic choice $\oplus \lambda$
- BUT! The denotation of the operator $\&$ is odd...
  - $\&$ isn’t idempotent.
Motivation: ProbNetKAT

- Syntactically, ProbNetKAT is a kind of Kleene algebra with probabilistic choice $\oplus_\lambda$
- BUT! The denotation of the operator $\&$ is odd...
  - $\&$ isn’t idempotent.
  - ; does not distribute over $\&$
Motivation: ProbNetKAT

- Syntactically, ProbNetKAT is a kind of Kleene algebra with probabilistic choice $\oplus_\lambda$
- BUT! The denotation of the operator $\&$ is odd...
  - $\&$ isn’t idempotent.
  - $;$ does not distribute over $\&$
- Why? What’s going on?
Motivation: ProbNetKAT

- Syntactically, ProbNetKAT is a kind of Kleene algebra with probabilistic choice $\oplus_\lambda$
- BUT! The denotation of the operator $\&$ is odd...
  - $\&$ isn’t idempotent.
  - $;$ does not distribute over $\&$
- Why? What’s going on?
- General question:

  *How can we add features in a principled and controllable manner?*
Building languages layer by layer: shopping list

Monad:

First layer:

\[ p ::= \text{skip} \mid p ; p \mid a \in \text{At} \](\text{At})\]skip](\text{At})\]skip = \text{skip} \mid p = p, \ldots \]

Monad:

Second layer:

\[ p ::= \text{abort} \mid p + p \mid a \in \text{At} \]

\[ p + \text{abort} = \text{abort} + p = p + p = p, \ldots \]

Monad:

Topping:

\[ p ::= p \oplus r p \mid a \in \text{At} \]

\[ p \oplus r q = q \oplus 1 - r p, \ldots \]
Building languages layer by layer: shopping list

- **First layer:**
  
  \[ p ::= \text{skip} \mid p; p \mid a \in \text{At} \]

  \[ p; \text{skip} = \text{skip}; p = p, \ldots \]

  Monad: \((-)^*\)
Building languages layer by layer: shopping list

Second layer:
\[ p ::= \text{abort} \mid p + p \mid a \in \text{At} \]
\[ p + \text{abort} = \text{abort} + p = p, \]
\[ p + q = q + p, p + p = p, \ldots \]
Monad: \( \mathcal{P} \)

First layer:
\[ p ::= \text{skip} \mid p ; p \mid a \in \text{At} \]
\[ p ; \text{skip} = \text{skip} ; p = p, \ldots \]
Monad: \( (\_\_\_)^* \)
Building languages layer by layer: shopping list

- **Topping:**
  \[ p ::= p \oplus_r p \mid a \in \text{At} \]
  \[ p \oplus_r q = q \oplus_1 \mathcal{I} p, \ldots \]
  Monad: \(\mathcal{D}\)

- **Second layer:**
  \[ p ::= \text{abort} \mid p + p \mid a \in \text{At} \]
  \[ p + \text{abort} = \text{abort} + p = p, \]
  \[ p + q = q + p, p + p = p, \ldots \]
  Monad: \(\mathcal{P}\)

- **First layer:**
  \[ p ::= \text{skip} \mid p;p \mid a \in \text{At} \]
  \[ p;\text{skip} = \text{skip};p = p, \ldots \]
  Monad: \((-\))^*\]
Combining the layers: things can go wrong!
Combining the layers: things can go wrong!

- The composition of two monads is not necessarily a monad
Combining the layers: things can go wrong!

- The composition of two monads is not necessarily a monad
- Combine monads $S$, $T$ via distributive law

$$\lambda : ST \rightarrow TS$$
Combining the layers: things can go wrong!

- The composition of two monads is not necessarily a monad
- Combine monads $S$, $T$ via distributive law
  \[
  \lambda : ST \to TS
  \]
- No distributive law $\mathcal{P}D \to \mathcal{D}P$
Combining the layers: things can go wrong!

- The composition of two monads is not necessarily a monad
- Combine monads $S$, $T$ via distributive law
  \[ \lambda : ST \to TS \]
- No distributive law $PD \to DP$
- But there exists a distributive law
  \[ (-)^*P \to P(-)^* \]
Combining the layers: things can go wrong!

- The composition of two monads is not necessarily a monad
- Combine monads $S$, $T$ via distributive law
  \[ \lambda : ST \to TS \]
- No distributive law $PD \to DP$
- But there exists a distributive law
  \[ (\neg)^*P \to P(\neg)^* \]
- How do we deal with this systematically?
This paper

A general and modular approach for determining:

(a) if a monad combination by distributive law is possible;
(b) if it is not possible, exactly which features are broken by the extension; and
(c) suggests a way to fix the composition by modifying one of the monads.
Monads

- Monads: a categorical way to encode computational effects:
Monads

- Monads: a categorical way to encode computational effects: Non-determinism, probabilities, side-effects...
Monads

- Monads: a categorical way to encode computational effects: Non-determinism, probabilities, side-effects ...  
- Applications of monads include programming language semantics, automata theory, etc.  
- It is convenient to compositionally combine several effects.
Definitions

Definition

A Monad \((T, \eta, \mu)\) on a category \(C\) is:

- An endofunctor \(T : C \to C\)
- A natural transformation \(\eta : 1 \to T\)
- A natural transformation \(\mu : TT \to T\)

(Verifying some structural properties)

We will consider monads on Set.
Examples

Finite Powerset\[ \mathcal{P}(A) = \{B \mid B \subseteq A, B \text{ finite}\} \]

Free Monoid (List)\[ A^* = \{w_1 \ldots w_n \mid n \in \mathbb{N}, w_i \in A\} \]

Distributions\[ \mathcal{D}(A) = \{f \mid f \text{ probability distribution on } A, \text{ and } \text{Supp}(f) \text{ finite}\} \]
Algebras

Definition

An *algebra* for the monad $T$ is an object $A$ together with a morphism $\alpha : TA \to A$.

(Verifying some structural properties involving $\eta$ and $\mu$)
Algebras

Definition

An *algebra* for the monad $T$ is an object $A$ together with a morphism $\alpha : TA \to A$.
(Verifying some structural properties involving $\eta$ and $\mu$)

Definition

For a signature $\Sigma$ and a set of equations $E$ we can define a monad $T$ such that $\mathbf{EM}(T) \simeq \mathbf{Alg}(\Sigma, E)$
Examples

<table>
<thead>
<tr>
<th>$\Sigma$</th>
<th>$E$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{P}$</td>
<td>0, +</td>
</tr>
<tr>
<td></td>
<td>x + 0 = 0 + x = x</td>
</tr>
<tr>
<td></td>
<td>x + y = y + x</td>
</tr>
<tr>
<td></td>
<td>(x + y) + z = x + (y + z)</td>
</tr>
<tr>
<td></td>
<td>x + x = x</td>
</tr>
<tr>
<td></td>
<td>(join-semilattice)</td>
</tr>
<tr>
<td>$(-)^*$</td>
<td>1, ;</td>
</tr>
<tr>
<td></td>
<td>x ; 1 = 1 ; x = x</td>
</tr>
<tr>
<td></td>
<td>(x ; y) ; z = x ; (y ; z)</td>
</tr>
<tr>
<td></td>
<td>(monoid)</td>
</tr>
</tbody>
</table>
$S, T$ monads, $\text{EM}(T) \simeq \text{Alg}(\Sigma T, E_T)$, $\text{EM}(S) \simeq \text{Alg}(\Sigma S, E_S)$

**Definition**

A *distributive law* of $T$ over $S$ is a natural transformation $\lambda : ST \rightarrow TS$ (verifying structural properties)
S, T monads, $\text{EM}(T) \simeq \text{Alg}(\Sigma_T, E_T)$, $\text{EM}(S) \simeq \text{Alg}(\Sigma_S, E_S)$

**Definition**

A *distributive law* of $T$ over $S$ is a natural transformation $\lambda : ST \to TS$ (verifying structural properties)

If $T$ distributes over $S$, then:

- $TS$ is a monad

- Operations in $\Sigma_S$ distribute over those of $\Sigma_T$

We call $S$ the *inner layer*, $T$ the *outer layer*. 
Remarks and questions:

- Distributive laws are one of the go-to methods to **compose monads**
- Implements a one-way distributivity of algebraic operations
- For two given monads, how to know whether there exists a distributive law?
- How to build it?
Remarks and questions:

- Distributive laws are one of the go-to methods to **compose monads**
- Implements a one-way distributivity of algebraic operations
- For two given monads, how to know whether there exists a distributive law?
- How to build it?

**Theorem**

Let $T$ be a monoidal monad, then for any finitary signature $\Sigma$, there exists a distributive law $\lambda_\Sigma : H_\Sigma T \to TH_\Sigma$ of the polynomial functor associated with $\Sigma$ over $T$. 
Remarks and questions:

- Distributive laws are one of the go-to methods to **compose monads**
- Implements a one-way distributivity of algebraic operations
- For two given monads, how to know whether there exists a distributive law?
- How to build it?

**Theorem**

Let $T$ be a monoidal monad, then for any finitary signature $\Sigma$, there exists a distributive law $\lambda_\Sigma : H_\Sigma T \rightarrow TH_\Sigma$ of the polynomial functor associated with $\Sigma$ over $T$.

Monoidal helps with lifting operations but not equations.
The procedure: 1. Build ‘candidate’ $\lambda : ST \to TS$
The procedure: 1. Build ‘candidate’ \( \lambda : ST \rightarrow TS \)

- \( S \) always given by signature \( \Sigma \) and equations \( E \)
The procedure: 1. Build ‘candidate’ \( \lambda : ST \to TS \)

- S always given by signature \( \Sigma \) and equations \( E \)
- Use monoidal ‘tensor’

\[ \otimes_{-,-} : T(-) \times T(-) \to T(- \times -) \]
The procedure: 1. Build ‘candidate’ $\lambda : ST \rightarrow TS$

- $S$ always given by signature $\Sigma$ and equations $E$
- Use monoidal ‘tensor’
  \[ \otimes_- : T(-) \times T(-) \rightarrow T(- \times -) \]
- Define a lifting $\hat{T}$ of $T$ on $\Sigma$-algebras
  \[ (A, \sigma : A^{\text{ar}(\sigma)} \rightarrow A)_{\sigma \in \Sigma} \rightarrow (TA, T\sigma \circ \otimes^{\text{ar}(\sigma)} : (TA)^{\text{ar}(\sigma)} \rightarrow TA)_{\sigma \in \Sigma} \]
The procedure: 1. Build ‘candidate’ $\lambda : ST \rightarrow TS$

- $S$ always given by signature $\Sigma$ and equations $E$
- Use monoidal ‘tensor’

\[
\otimes_{-, -} : T(-) \times T(-) \rightarrow T(- \times -)
\]

- Define a lifting $\hat{T}$ of $T$ on $\Sigma$-algebras

\[
(A, \sigma : A^{\text{ar}(\sigma)} \rightarrow A)_{\sigma \in \Sigma} \rightarrow (TA, T \sigma \circ \otimes^{\text{ar}(\sigma)} : (TA)^{\text{ar}(\sigma)} \rightarrow TA)_{\sigma \in \Sigma}
\]

- $\hat{;} : (P(A^*)^2 \rightarrow P(A^*), (U, V) \mapsto \{u; v \mid u \in U, v \in V\}, \text{skip} = \{\epsilon\}$
The procedure: 1. Build ‘candidate’ \( \lambda : ST \to TS \)

- \( S \) always given by signature \( \Sigma \) and equations \( E \)
- Use monoidal ‘tensor’
  \[
  \otimes_-, - : T(-) \times T(-) \to T(- \times -)
  \]
- Define a lifting \( \hat{T} \) of \( T \) on \( \Sigma \)-algebras
  \[
  (A, \sigma : A^{\text{ar}(\sigma)} \to A)_{\sigma \in \Sigma} \to (TA, T\sigma \circ \otimes^{\text{ar}(\sigma)} : (TA)^{\text{ar}(\sigma)} \to TA)_{\sigma \in \Sigma}
  \]
- \( \hat{;} : (\mathcal{P}(\text{At}^*))^2 \to \mathcal{P}(\text{At}^*), (U, V) \mapsto \{u; v \mid u \in U, v \in V\}, \text{skip} = \{\epsilon\} \)

**Theorem**

If \( \hat{T} \) sends \((\Sigma, E)\)-algebras to \((\Sigma, E)\)-algebras, then it defines a distributive law \( \lambda : ST \to TS \).
The procedure: 2. Check if $\hat{T} : \text{Alg}(\Sigma, E) \rightarrow \text{Alg}(\Sigma, E)$
The procedure: 2. Check if $\hat{T} : \text{Alg}(\Sigma, E) \rightarrow \text{Alg}(\Sigma, E)$

Illustration with idempotency, $(A, \bullet : A^2 \rightarrow A) \models x \bullet x = x$
The procedure: 2. Check if $\hat{T} : \text{Alg}(\Sigma, E) \to \text{Alg}(\Sigma, E)$
Illustration with idempotency, $(A, \cdot : A^2 \to A) \models x \cdot x = x$
The procedure: 2. Check if \( \hat{T} : \text{Alg}(\Sigma, E) \to \text{Alg}(\Sigma, E) \)

Illustration with idempotency, \((A, \cdot : A^2 \to A) \models x \cdot x = x\)

\[
\begin{array}{ccc}
TA & \xrightarrow{T\Delta} & T(A^2) \\
\downarrow \text{Id}_A & & \downarrow T\cdot \\
TA & & TA
\end{array}
\]
The procedure: 2. Check if $\hat{T} : \text{Alg}(\Sigma, E) \to \text{Alg}(\Sigma, E)$

Illustration with idempotency, $(A, \bullet : A^2 \to A) \models x \bullet x = x$
The procedure: 2. Check if $\hat{T} : \text{Alg}(\Sigma, E) \rightarrow \text{Alg}(\Sigma, E)$

Illustration with idempotency, $(A, \bullet : A^2 \rightarrow A) \models x \bullet x = x$

We call the upper triangle the *residual diagram* of $x \bullet x = x$. 
The procedure: 2. Check if \( \hat{T} : \mathbf{Alg}(\Sigma, E) \to \mathbf{Alg}(\Sigma, E) \)

Illustration with idempotency, \((A, \cdot : A^2 \to A) \models x \cdot x = x\)

We call the upper triangle the \textit{residual diagram} of \( x \cdot x = x \).

If it commutes then \((TA, \hat{\cdot} : (TA)^2 \to TA) \models x \hat{\cdot} x = x\).
The procedure: 2. Check if $\hat{T} : \text{Alg}(\Sigma, E) \to \text{Alg}(\Sigma, E)$

Illustration with idempotency, $(A, \bullet : A^2 \to A) \models x \bullet x = x$

We call the upper triangle the *residual diagram* of $x \bullet x = x$.

If it commutes then $(TA, \hat{\bullet} : (TA)^2 \to TA) \models x \hat{\bullet} x = x$.

If it doesn’t, we know exactly where the obstacle is and can troubleshoot accordingly.
Application: combining $(-)^*$, $\mathcal{P}$ and $\mathcal{D}$

- $\mathcal{P}$ sends monoids to monoids $\Rightarrow \mathcal{P}(-)^*$ is a monad: the free idempotent semiring monad.
Application: combining \((-\)^*, \mathcal{P}\) and \mathcal{D}

- \mathcal{P} sends monoids to monoids \(\Rightarrow \mathcal{P}(-)^*\) is a monad: the free idempotent semiring monad.
- Distributivity and absorption law are enforced by design!
Application: combining \((-\))^*, \mathcal{P} and \mathcal{D}

- \mathcal{P} sends monoids to monoids \(\Rightarrow \mathcal{P}(-)^*\) is a monad: the free idempotent semiring monad.
- Distributivity and absorption law are enforced by design!
- \(\mathcal{D}\) with \(\otimes\) given by the product measure makes the residual diagrams for associativity, units and commutativity commute...
Application: combining $(-)^*$, $\mathcal{P}$ and $\mathcal{D}$

- $\mathcal{P}$ sends monoids to monoids $\Rightarrow \mathcal{P}(-)^*$ is a monad: the free idempotent semiring monad.
- Distributivity and absorption law are enforced by design!
- $\mathcal{D}$ with $\otimes$ given by the product measure makes the residual diagrams for associativity, units and commutativity commute...
- but neither for IDEMPOTENCY, nor for DISTRIBUTIVITY!
Application: combining \((-\)^*, \mathcal{P} and \mathcal{D}

- \mathcal{P} sends monoids to monoids \(\Rightarrow \mathcal{P}(\mathcal{-})^*\) is a monad: the free idempotent semiring monad.
- Distributivity and absorption law are enforced by design!
- \(\mathcal{D}\) with \(\otimes\) given by the product measure makes the residual diagrams for associativity, units and commutativity commute...
- but neither for IDEMPOTENCY, nor for DISTRIBUTIVITY!
- Troubleshooting: remove those axioms
Application: combining \((-\))^*, \mathcal{P} and \mathcal{D}

- \mathcal{P} sends monoids to monoids \implies \mathcal{P}(-)^* is a monad: the free idempotent semiring monad.
- Distributivity and absorption law are enforced by design!
- \mathcal{D} with \otimes given by the product measure makes the residual diagrams for associativity, units and commutativity commute...
- but neither for IDEMPOTENCY, nor for DISTRIBUTIVITY!
- Troubleshooting: remove those axioms
- There IS a distributive law over \mathcal{D} of the monad defined by

\[
\begin{align*}
\text{p; skip} &= \text{skip; p} = \text{p}, & (\text{p; q); r} &= \text{p}(\text{q; r}), \\
\text{p + abort} &= \text{abort} + \text{p} = \text{p}, & \text{p + q} &= \text{q} + \text{p}, & (\text{p + q}) + \text{r} &= \text{p} + (\text{q + r}), \\
\text{p; abort} &= \text{abort; p} = \text{abort}
\end{align*}
\]

Completely consistent with the semantics of ProbNetKAT
Application: combining \((-\))^*, \mathcal{P} and \mathcal{D}

- \mathcal{P} sends monoids to monoids \(\Rightarrow \mathcal{P}(-)^*\) is a monad: the free idempotent semiring monad.
- Distributivity and absorption law are enforced by design!
- \mathcal{D} with \(\otimes\) given by the product measure makes the residual diagrams for associativity, units and commutativity commute...
- but neither for IDEMPOTENCY, nor for DISTRIBUTIVITY!
- Troubleshooting: remove those axioms
- There IS a distributive law over \(\mathcal{D}\) of the monad defined by

\[
\begin{align*}
p; \text{skip} &= \text{skip}; p = p, \quad (p; q); r &= p(q; r), \\
p + \text{abort} &= \text{abort} + p = p, \quad p + q &= q + p, \quad (p + q) + r &= p + (q + r), \\
p; \text{abort} &= \text{abort}; p = \text{abort}
\end{align*}
\]

- Completely consistent with the semantics of ProbNetKAT
Theorem
Let T be a commutative, relevant and affine monad. For all u and v, T preserves $u = v$. 
Fixing composition – Method 1: changing the inner layer

Idea: remove the faulty laws from the inner layer.
Fixing composition – Method 1: changing the inner layer

Idea: remove the faulty laws from the inner layer.

\( \text{EM}(S) \simeq \text{Alg}(\Sigma_S, E_S), \text{EM}(T) \simeq \text{Alg}(\Sigma_T, E_T) \). Let \( E'_S \) be the subset of \( E_S \) containing the equations preserved by \( T \).

- Obtain \( S' \) from \( \text{Alg}(\Sigma_S, E'_S) \)
- Compose \( T \) with \( S' \), obtain a \( (\Sigma, E) \) algebra, where:

\[
\Sigma = (\Sigma_T \cup \Sigma_S)
\]

\[
E = (E_T \cup E'_S \cup \text{distributivity of } \Sigma_S \text{ over } \Sigma_T)
\]
Method 1: fix the inner layer

Example

\( \mathcal{D} \) does not preserve idempotency nor distributivity. Drop them and obtain a \((\Sigma, E)\) algebra where \( \Sigma = \{; , 1, +, 0, \oplus_\lambda \} \) and \( E = \)

- associativity, commutativity, unit laws for +
- equations of \((-)^*\)
- absorption \( p; 0 = 0; p = 0 \)
- equations of \( \mathcal{D} \) (convex algebras)
  - \( p; (q \oplus_\lambda r) = (p; q) \oplus_\lambda (p; r) \)
  - \( (q \oplus_\lambda r); p = (q; p) \oplus_\lambda (r; p) \)
  - \( p + (q \oplus_\lambda r) = (p + q) \oplus_\lambda (p + r) \)
  - \( (q \oplus_\lambda r) + p = (q + p) \oplus_\lambda (r + p) \)
Method 2: Change the outer layer

Idea: consider the largest submonad preserving the faulty equations.
Method 2: Change the outer layer

Idea: consider the largest submonad preserving the faulty equations.

Example

$\mathcal{PD}$ is not a monad as $\mathcal{P}$ does not preserve idempotency. The largest submonad of $\mathcal{P}$ preserving it is the *convex powerset* $\mathcal{P}_c$. 
Method 2: Change the outer layer

Idea: consider the largest submonad preserving the faulty equations.

Example

$\mathcal{PD}$ is not a monad as $\mathcal{P}$ does not preserve idempotency. The largest submonad of $\mathcal{P}$ preserving it is the \textit{convex powerset} $\mathcal{P}_c$.

Two options to fix $\mathcal{PD}$:

1. Build a monad $PD$ that preserves the relevant equations.
2. Replace $\mathcal{P}$ by $\mathcal{P}_c$ and then composition works: $\mathcal{P}_c \mathcal{D}$. 
Conclusions

- A principled approach to constructing (equational) languages layer by layer.
- Conditions on existence of distributive laws and potential fixing strategies.
- Note: other troubleshooting strategies are possible!
Thank you! Questions?