Coalgebra, lecture 11: Distributive Laws

Jurriaan Rot (and Aleks Kissinger)

November 22, 2016

1 Operations on stream systems

We start with everbody's favorite operation on streams, $\mathsf{alt}\colon A^\omega \times A^\omega \to A^\omega$, defined by stream differential equations:

$$\mathsf{alt}(\sigma,\tau)(0) = \sigma(0) \qquad \mathsf{alt}(\sigma,\tau)' = \mathsf{alt}(\tau',\sigma') \tag{1}$$

These equations uniquely define alt. One way to see that, is by turning the domain $A^\omega \times A^\omega$ into a stream system, in such a way that alt arises as the unique homomorphism to the final coalgebra A^ω. We're going to look at such definitions a bit more systematically and generally.

First, we rephrase the definition of alt as the following inference rule (or actually rules, one for each $a, b \in A$):

$$\frac{x \xrightarrow{a} x' \qquad y \xrightarrow{a} y'}{\mathsf{alt}(x,y) \xrightarrow{a} \mathsf{alt}(y',x')} \tag{2}$$

What do we mean with these rules? The idea is that they define alt as a construction on stream systems. More precisely, let $\langle o, f\rangle\colon X \to A \times X$ be a stream system. If we interpret $x \xrightarrow{a} x'$ as $\langle o, f\rangle(x) = (a, x')$, then the above rules give us a new stream system, defined by:

$$\begin{aligned} \mathsf{alt}_f\colon X \times X &\to A \times (X \times X) \\ (x,y) &\mapsto (o(x), (f(y), f(x))) \end{aligned}$$

If we apply this construction to the *final* coalgebra $z\colon A^\omega \to A \times A^\omega$, we obtain alt, as in (1), as the unique homomorphism to the final coalgebra.

$$\begin{array}{ccc} A^\omega \times A^\omega & \overset{\mathsf{alt}}{- - \to} & A^\omega \\ \downarrow{\scriptstyle \mathsf{alt}_z} & & \downarrow{\scriptstyle z} \\ A \times (A^\omega \times A^\omega) & \xrightarrow[\mathsf{id}_A \times \mathsf{alt}]{} & A \times A^\omega \end{array} \tag{3}$$

We make all of this a bit more precise. What we have here is:

- *syntax*: one binary operation alt, modeled by the functor $S\colon \mathsf{Set} \to \mathsf{Set}$, $S(X) = X \times X$;
- *behaviour*: stream systems, modeled by (coalgebras for) the well-known functor $B\colon \mathsf{Set} \to \mathsf{Set}$, $B(X) = A \times X$;
- the *specification* of alt, as given in (2). We model it by a *distributive law*, which is a natural transformation $\lambda\colon SB \Rightarrow BS$, in this case defined as follows:

$$\begin{aligned} \lambda_X &\colon (A \times X) \times (A \times X) \to A \times (X \times X) \\ &((a, x'), (b, y')) \mapsto (a, (y', x')) \end{aligned} \tag{4}$$

This allows us to map a stream system $X \xrightarrow{f} A \times X$ to

$$\mathsf{alt}_f\colon X \times X \xrightarrow{f} (A \times X) \times (A \times X) \xrightarrow{\lambda_X} A \times (X \times X)\,.$$

The point here is that λ here fully captures the specification of alt, in a precise way, which we can immedately apply to any stream system. In particular, applying it to the final coalgebra, we get alt_z and alt as in (3).

2 Distributive laws

The above story about alt is, of course, an instance of a much more general picture. This picture involves a functor $S\colon \mathcal{C} \to \mathcal{C}$ on a category $\mathcal{C}$ modelling (for now) the syntax of some operation(s) we would like to define (such as a binary operator, or a whole collection of operators); $B\colon \mathcal{C} \to \mathcal{C}$ modelling the behaviour of interest (such as streams, transition systems, or automata); and a distributive law between them, modelling a specification of the operations (such as the specification of alt).

Definition 2.1. Let $S, B\colon \mathcal{C} \to \mathcal{C}$ be functors. A *distributive law* of S over B is a natural transformation of the form $\lambda\colon SB \Rightarrow BS$.

Such a distributive law defines a functor $\overline{S}\colon \mathsf{Coalg}(B) \to \mathsf{Coalg}(B)$, defined on objects by:

$$X \xrightarrow{f} BX \quad \mapsto \quad SX \xrightarrow{Sf} SBX \xrightarrow{\lambda_X} BSX\,.$$

and on arrows by $\overline{S}(h) = S(h)$. It is a nice exercise to prove that this is well-defined, i.e., that $\overline{S}(h)$ is indeed a coalgebra homomorphism. Notice that, if we apply $\overline{S}$ to a coalgebra with carrier X, then the result is a coalgebra with carrier $S(X)$. A fancy way of saying this is that $\overline{S}$ is a *lifting* of S, which means that the following diagram of categories and functors commutes:

$$\begin{array}{ccc} \mathsf{Coalg}(B) & \xrightarrow{\overline{S}} & \mathsf{Coalg}(B) \\ {\scriptstyle U}\downarrow & & \downarrow{\scriptstyle U} \\ \mathcal{C} & \xrightarrow[S]{} & \mathcal{C} \end{array}$$

where U is the forgetful functor, mapping a coalgebra (X, f) to its carrier X.

Now, if B has a final coalgebra (Z, z), then we can apply the lifting $\overline{S}$ (the result is on the left-hand side of the diagram below) to it and obtain a homomorphism from SZ to Z, which we'll call beh^λ:

$$\begin{array}{ccc} SZ & \overset{\mathsf{beh}_\lambda}{\dashrightarrow} & Z \\ {\scriptstyle Sz}\downarrow & & \\ SBZ & & \downarrow{\scriptstyle z} \\ {\scriptstyle \lambda_Z}\downarrow & & \\ BSZ & \xrightarrow[B(\mathsf{beh}_\lambda)]{} & BZ \end{array} \tag{5}$$

Notice that beh^λ is an *algebra* for the functor S (which in fact is an algebra for $\overline{S}$). So a distributive law apparently always defines an *algebra on the final coalgebra*.

If we think of λ as a specification of operations, then we can think of beh_λ as the interpretation of these operations on the final coalgebra. For instance:

- In the setting of the previous section, if we define λ as in (4), $\mathsf{beh}_\lambda \colon A^\omega \times A^\omega$ becomes the alt operation on A^ω.
- We can define multiple operations at once, simply with a different choice of S. For instance, to combine alt with a constant stream a for every $a \in A$, we take $S(X) = A + (X \times X)$, and define $\lambda \colon SB \Rightarrow BS$ by

$$\begin{aligned} \lambda_X \colon A + ((A \times X) \times (A \times X)) &\to A \times (A + (X \times X)) \\ (a, 1) &\mapsto (a, (a, 1)) \\ (((a, x'), (b, y')), 2) &\mapsto (a, ((y', x'), 2)) \end{aligned}$$

- Let $B(X) = 2 \times X^A$. We will define another binary operation, so let $S(X) = X \times X$. Consider $\lambda_X \colon SB \Rightarrow BX$, defined by

$$\begin{aligned} \lambda_X \colon (2 \times X^A) \times (2 \times X^A) &\to 2 \times (X \times X)^A \\ ((b, f), (c, g)) &\mapsto (b \wedge c, a \mapsto (f(a), g(a))) \end{aligned}$$

  Written in terms of rules, we are defining a binary operation $\otimes$:

$$\frac{x \xrightarrow{a} x' \quad y \xrightarrow{a} y'}{x \otimes y \xrightarrow{a} x' \otimes y'} \qquad \frac{x \downarrow \quad y \downarrow}{(x \otimes y) \downarrow}$$

  The (carrier of the) final coalgebra of B is $\mathcal{P}(A^*)$. What will $\mathsf{beh}_\lambda \colon \mathcal{P}(A^*) \times \mathcal{P}(A^*) \to \mathcal{P}(A^*)$ be?

2.1 What operations can we define?

The main point so far of using distributive laws, is that they give us an abstract specification format for operations on (final) coalgebras. We have used that to define operations on streams and automata, but you could plug in any B instead. But one can wonder how much the specification formats allow us to do. For instance, consider the rule

$$\frac{x \xrightarrow{a} x'}{f(x) \xrightarrow{a} f(f(x')))}$$

which defines a (silly) operation on stream systems. If we try to model that as $\lambda\colon SB \Rightarrow BS$, with $S(X) = X$ and $B(X) = A \times X$, we get stuck: for the right-hand side $f(f(x))$ we need "two levels" of S.

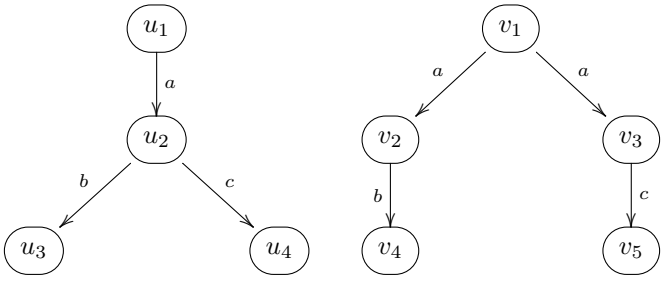
The solution is to look instead at different types of natural transformations. In particular, natural transformations of the form $\lambda\colon SB \Rightarrow BS^*$, where S^* is the free monad for S, do allow to write rules such as the above. They induce (and correspond, in a sense that is beyond the current treatment) distributive laws of the form $\rho\colon S^*B \Rightarrow BS^*$, essentially by induction. With an even more general version we can define, for instance, the operations of regular expressions as natural transformations. But we won't go into this for now.

3 Determinisation

In one of the previous lectures, we've looked at non-deterministic automata. There can be modeled as coalgebras: a non-deterministic automaton is a coalgebra of the form

$$\langle \epsilon, \delta \rangle\colon X \to 2 \times (\mathcal{P}(X))^A$$

(we might as well take the finite powerset functor, if we'd like to speak about a final coalgebra, but we don't really need to at this point). The associated coalgebraic notion of bisimilarity does not coincide with language equivalence. The classical example is the automata below:



(6)

States u_1 and v_1 are clearly language equivalent, both accepting the empty language, but they are not bisimilar. This is not wrong: the branching aspect is something we can actually observe, and in some case it makes sense not to equate the above systems.

However, in many cases we're actually interested in language equivalence, and of course we'd like to fit that into the coalgebraic picture. To this end, we'll use a classic trick: the powerset construction, which turns a non-deterministic automaton into a deterministic automaton. In fact—like many things in the theory of coalgebras—this turns out to be yet another instance of something general and abstract, involving distributive laws.

First, the concrete definition. From a given non-deterministic automaton $\langle \epsilon, \delta, \rangle \colon X \to 2 \times (\mathcal{P}(X))^A$, we define $\langle \epsilon^\sharp, \delta^\sharp \rangle \colon \mathcal{P}(X) \to 2 \times (\mathcal{P}(X))^A$ by

$$\epsilon^\sharp(S) = \bigvee_{x \in S} \epsilon(x)$$
$$\delta^\sharp(S)(a) = \bigcup_{x \in S} \delta(s)(a)$$

for all $S \in \mathcal{P}(X)$ and $a \in A$. This is a *deterministic automaton*, whose carrier is the powerset of states. This process is exactly the classical powerset construction. In a sense, we're hiding the stuff we'd like not to observe inside the state space.

We obtain a language semantics $\mathsf{beh} \colon \mathcal{P}(X) \to \mathcal{P}(A^*)$ by finality. This assigns to each set $S \in \mathcal{P}(X)$ a language; we obtain the language of a state $x \in X$ by applying it to the singleton $\{x\}$, i.e., $\mathsf{beh}(\{x\})$. All in all, we get the following picture:

$$\begin{array}{ccccc} X & \xrightarrow{\eta_X} & \mathcal{P}(X) & \xrightarrow{\mathsf{beh}} & \mathcal{P}(A^*) \\ {\scriptstyle \langle \epsilon, \delta \rangle} \downarrow & \swarrow_{\langle \epsilon^\sharp, \delta^\sharp \rangle} & & & \downarrow \\ 2 \times (\mathcal{P}(X))^A & & \xrightarrow[\mathsf{id} \times (\mathsf{beh})^A]{} & & 2 \times \mathcal{P}(A^*)^A \end{array} \tag{7}$$

The arrow on the right is the final coalgebra of languages. The $\eta_X \colon X \to \mathcal{P}(X)$ just maps each element of X to the corresponding singleton. If you did the exercises last week, you'll recognise it as the X-component of the unit of the powerset monad—this will be important later on. The *language semantics* now arises as the composite $\mathsf{beh} \circ \eta$. That this is really the intended language semantics is just the correctness of the powerset construction.

It turns out this whole construction is actually concisely captured by a distributive law. First, we decompose the functor of non-deterministic automata as $B\mathcal{P}$, where $BX = 2 \times X^A$ (the functor for deterministic automata) and $\mathcal{P}$ is the powerset functor (giving the branching/non-determinism). Now, the distributive law is $\lambda \colon \mathcal{P}B \Rightarrow B\mathcal{P}$, given on a component X by:

$$\lambda_X \colon \mathcal{P}(2 \times X^A) \to 2 \times (\mathcal{P}(X))^A$$
$$S \mapsto \Big(\bigvee_{(b,f) \in S} b, a \mapsto \bigcup_{(b,f) \in S} f(a) \Big)$$

This distributive law takes a *set* of "behaviours of a deterministic automaton", and turns it into a "single" (non-deterministic) behaviour. How does this encode

the powerset construction? Well, a little thought (and/or computation) shows us that the automaton in (8) correponds to:

$$\mathcal{P}(X) \xrightarrow{\mathcal{P}(\langle\epsilon,\delta\rangle)} \mathcal{P}(2 \times (\mathcal{P}(X))^A) \xrightarrow{\lambda_{\mathcal{P}(X)}} 2 \times (\mathcal{P}(\mathcal{P}(X)))^A \xrightarrow{\mathsf{id}\times(\mu_X)^A} 2 \times (\mathcal{P}(X))^A$$

Here $\mu_X \colon \mathcal{P}(\mathcal{P}(X)) \to \mathcal{P}(X)$ is given by union; it is the multiplication of the powerset monad $(\mathcal{P}, \eta, \mu)$.

With this distributive law, its not so difficult to prove that we obtain a functor

$$\overline{\mathcal{P}} \colon \mathsf{Coalg}(B\mathcal{P}) \to \mathsf{Coalg}(\mathcal{P})$$

from the category of non-deterministic automata to deterministic automata, defined on objects by the powerset construction. Actually, this is another example of a *lifting* defined by a distributive law, although it is different than the one we've seen before.

$$\begin{array}{ccc} \mathsf{Coalg}(B\mathcal{P}) & \xrightarrow{\overline{\mathcal{P}}} & \mathsf{Coalg}(B) \\ {\scriptstyle U}\downarrow & & \downarrow{\scriptstyle U} \\ \mathsf{Set} & \xrightarrow[\mathcal{P}]{} & \mathsf{Set} \end{array}$$

The fact that it is a lifting, means that it acts as the powerset on the state space of automata (and on homomorphisms).

Let's see a different example. A *partial automaton* is a coalgebra of the form $\langle\epsilon, \delta\rangle \colon X \to 2 \times (X + 1)^A$, where $1 = \{*\}$. For every state and alphabet symbol, we have either a single next state, or no next state. The latter should just mean that no more words with that letter in front should be accepted. Again, this is not captured by coalgebraic bisimilarity, which sees the 1 part as explicit termination (a nice exercise to work out).

So we'll use another determinisation procedure. Now, we construct a deterministic automaton $\langle\epsilon^\sharp, \delta^\sharp\rangle \colon X + 1 \to 2 \times (X + 1)^A$ (notice that it's state space is $X + 1$) defined as follows: ... (left as an exercise!).

The idea is that this construction canonically adds a "sink" or "trap" state, which is not accepting and has transitions to itself for every alphabet letter. Since the result is a deterministic automaton, we get a unique coalgebra morphism $\mathsf{beh} \colon X + 1 \to \mathcal{P}(A^*)$. And the language of a state $x \in X$ is just the language $\mathsf{beh}(x, 1)$. We get another nice picture.

$$\begin{array}{ccccc} X & \xrightarrow{\eta_X} & X+1 & \xrightarrow{\mathsf{beh}} & \mathcal{P}(A^*) \\ {\scriptstyle \langle\epsilon,\delta\rangle}\downarrow & \swarrow{\scriptstyle \langle\epsilon^\sharp,\delta^\sharp\rangle} & & & \downarrow \\ 2 \times (X+1)^A & & \xrightarrow[\mathsf{id}\times(\mathsf{beh})^A]{} & & 2 \times \mathcal{P}(A^*)^A \end{array} \tag{8}$$

The η_X here maps x to $(x, 1)$. The *language semantics* arises as the composite $\mathsf{beh} \circ \eta$.

Last week, we've seen the Option monad, which we'll call (O, η, μ) here to save some typing work for the author. Partial automata are coalgebras for the composite functor BO. Similar to the case of non-deterministic automata, the determinisation construction for partial automata is captured concisely as a distributive law $\lambda\colon OB \Rightarrow BO$. And again, the determinisation construction can be rephrased in terms of λ. All of this is left as an exercise.

4 Determinisation: the general picture

We have seen two determinisation constructions, for non-deterministic automata and partial automata. In both cases, the main ingredients were:

1. a functor B, whose coalgebras model deterministic automata;
2. a monad (T, η, μ), which captures the branching aspect that we would like to hide;
3. a distributive law $\lambda\colon TB \Rightarrow BT$, which captures the actual determinisation construction.

So both seem to be part of a general picture. Indeed, with the above ingredients, for an arbitrary coalgebra $f\colon X \to BTX$, we can define a B-coalgebra

$$TX \xrightarrow{Tf} TBTX \xrightarrow{\lambda_X} BTTX \xrightarrow{B\mu_X} BTX \tag{9}$$

which is the "determinisation" of f. This construction defines a lifting of T:

$$\begin{array}{ccc} \mathsf{Coalg}(BT) & \xrightarrow{\overline{T}} & \mathsf{Coalg}(B) \\ {\scriptstyle U}\downarrow & & \downarrow{\scriptstyle U} \\ \mathcal{C} & \xrightarrow[T]{} & \mathcal{C} \end{array}$$

giving an abstract determinisation procedure. This picture generalizes both the one for non-deterministic automata and for partial automata; and there are many other examples.

There is a lot more to say about this, in particular on the interplay between the distributive law and the monad structure. If that sounds exciting to you, have a look at the paper below.

References

[1] Alexandra Silva, Filippo Bonchi, Marcello M. Bonsangue, Jan J. M. M. Rutten: Generalizing determinization from automata to coalgebras. Logical Methods in Computer Science 9(1) (2013)