Coalgebraic Semantics

Lecture 4: CoCaml is a programming language. We live in a society.

Hakan Dingenc
hakan@u.northwestern.edu

Pedro Amorim
pamorim@cs.cornell.edu

Siva Somayyajula
ssomayya@cs.cmu.edu

June 20, 2019

Remark. If we have a final coalgebra (Z, α) and any other coalgebra (X, β) over a functor F, which intuitively provides an *operational semantics* for sets X, Y, etc. and functions α, β, etc., then the induced map $h : X \to Z$ computes the *denotation* of syntactic objects in X. Furthermore, $F(h)$ is the denotational semantics viewed operationally. The following diagram summarizes the setup:

$$\begin{array}{ccc} X & \xrightarrow{\quad h \quad} & Z \\ {\scriptstyle\beta}\downarrow & & \downarrow{\scriptstyle\alpha} \\ F(X) & \xrightarrow[F(h)]{} & F(Z) \end{array}$$

Definition 1 (Regular expression)**.** $\mathrm{Reg} \ni r := 0 \mid 1 \mid r + r \mid r; r \mid r^*$.

Theorem 1. *The final coalgebra of $X \mapsto \mathbf{2} \times X^A$ is carried by 2^{A^*} i.e. the set of acceptors on A^*.*

Proof. Let $\mathbf{2}^{A^*} \xrightarrow{\langle \epsilon?, \delta \rangle} \mathbf{2} \times \left(\mathbf{2}^{A^*}\right)^A$ be given by $\epsilon?(L) = [\epsilon \in L]$ and $\delta(L)(a)(u) = [au \in L]$ where $[\cdot]$ is the Iverson bracket. For any $Y \xrightarrow{\langle o, t \rangle} \mathbf{2} \times Y^A$, let $h(x)(\epsilon) = o(x)$ and $h(x)(au) = h(t(x)(a))(u)$. Uniqueness follows by induction on the input, whose principle can be established in a similar fashion to that of the natural numbers. □

Example 1. The above definition allows us to give functorial semantics to regular expressions. By defining the *Brzozowski derivatives* $\mathrm{Reg} \xrightarrow{\langle E, D \rangle} \mathbf{2} \times \mathrm{Reg}^A$, the induced map $\mathrm{Reg} \xrightarrow{[\![\cdot]\!]} \mathbf{2}^{A^*}$ is the standard interpretation of regular expressions by their corresponding acceptors. Intuitively, E determines whether its input accepts the empty string i.e. $E(r) \Leftrightarrow \epsilon \in [\![r]\!]$. Then, D calculates the "residual" regular expression of its input after observing the character a i.e. $[\![r]\!] = \{au \mid u \in [\![D_a(r)]\!]\}$. From this specification, their definitions follow.

	E	D_a
0	0	0
1	1	0
b	0	$[a = b]$
$r + s$	$E(r) \vee E(s)$	$D_a(r) + D_a(s)$
$r; s$	$E(r) \wedge E(s)$	$D_a(r); s + E(r); D_a(s)$
r^*	1	$D_a(r); r^*$

As expected, one can show the following, where ; lifts concatenation to sets of strings and $*$ denotes

Kleene closure. Note that we identify acceptors with the set of strings they accept.

$$\begin{aligned}
[\![0]\!] &= \emptyset \\
[\![1]\!] &= A^* \\
[\![b]\!] &= \{b\} \\
[\![r;s]\!] &= [\![r]\!];[\![s]\!] \\
[\![r+s]\!] &= [\![r]\!] \cup [\![s]\!] \\
[\![r^*]\!] &= [\![r]\!]^*
\end{aligned}$$

Example 2. By negating E, we yield the interpretation of regular expressions as the *complement* languages they denote.

Example 3. $(\mathrm{Reg}, [0, 1, a, ;, +])$ (which is an isomorphism) is the initial algebra on $X \mapsto \mathbf{1} + \mathbf{1} + A + X \times X + X \times X$. For any algebra $\mathbf{1} + \mathbf{1} + A + Y \times Y + Y \times Y \xrightarrow{[o,i,b,\cdot,\oplus]} Y$, the induced map $h : \mathrm{Reg} \to Y$ is given by $h(0) = o$, $h(1) = i$, $h(a) = b$, $h(r;s) = h(r) \cdot h(s)$, $h(r \oplus s) = h(r) \oplus h(s)$, and $h(r^*) = h(r) \cdot h(r^*)$. Now, define an algebra on 2^{A^*} over S mirroring the definition of $[\![\cdot]\!]$ above component-wise. One can show that the induced map of type $\mathrm{Reg} \to 2^{A^*}$ is once again the standard interpretation.

Remark. There is a *natural transformation* of type $F \circ S^* \Rightarrow S^* \circ F$[1] that determines the distributive laws on regular expressions.

Theorem 2. *Given a general definition of coalgebraic bisimulation, for a large class of functors F, $X \sim_F X' \iff [\![X]\!] = [\![X']\!]$, allowing the question of program equivalence to be reduced to constructing a bisimulation.*

Remark. With bialgebras, which have algebraic and coalgebraic structure, one may use bisimulation and the equational reasoning afforded by algebraicity to execute brutal power moves. For example, it is common to define the automaton corresponding to a particular regular expression where states are residual regular expressions, the transitions are computed by D, and the output function is given by E. One may use bisimulation to show language equivalence but algebraicity to perform equational identification on states.

Remark. In bialgebraic semantics with functors S and F the syntax of the language is defined as the initial S-algebra (remember how inductively defined ASTs can be seen as initial algebras). Then by defining an F-coalgebra $\alpha : \mathsf{AST} \to F(\mathsf{AST})$ we're fixing a specific operational semantics over the syntax. Assuming that Z is the final F, the denotational semantics $[\![\cdot]\!] : \mathsf{AST} \to Z$ is defined as the unique morphism given by finality. Theorem 2 states that the semantics is fully abstract and contextually equivalent.

Remark. Depending on the context we may need different base categories to define our semantics. Some examples are using nominal sets for languages with name binding and convex sets for probabilistic languages. See "Enhanced Coinduction" by Jurriaan Rot.

CoCaml

In OCaml it's possible to define coinductive objects. The infinite list $(1, 2, 1, 2...)$ is defined in OCaml by code below:

```
let rec alt = 1 :: 2 :: alt
```

However, due to how OCaml computes fixpoints, valid recursive programs on codata diverges. To mitigate this, CoCaml offers alternative fixpoint computations.

[1] To deal with the Kleene star we need a different functor S^*, instead of S. Since there are some order-theoretic questions that need to be addressed in order to show that the algebra structure is well-defined, we allude S^*'s full definition and leave it to an interested reader.

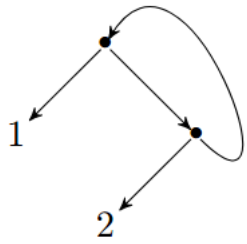

Figure 1: Infinite list

For instance, a program that computes the elements of infinite lists should converge for the list above. In this particular case, such a function is trying to compute the following fixpoint equation:

$$\begin{aligned} \mathsf{elements}\ alt &= \mathsf{insert}\ 1\ (\mathsf{elements}(2 :: alt)) \\ \mathsf{elements}\ (2 :: alt) &= \mathsf{insert}\ 2\ (\mathsf{elements}(alt)) \end{aligned}$$

In OCaml, such a fixpoint would be computed by unfolding elements alt until it converges. Since alt is infinite, this unfolding diverges. However, there are other methods to compute fixpoints. CoCaml implements a few of them and let's you choose which one to use. See the original paper for more details[2].

[2] `http://www.cs.cornell.edu/~kozen/Papers/CoCaml.pdf`