Coalgebra, lecture 12: Lattices and Coinduction

Jurriaan Rot

November 28, 2016

So far, we've talked quite a bit about coinduction within the framework of (final) coalgebras. In this lecture, we will look at a more basic foundation, in terms of *lattices*, which are certain ordered structures. This lattice-theoretic perspective on coinduction is, as we will see later, a special case of the theory of coalgebras; however, it is interesting on its own for a variety of reasons. We'll start with a few examples, and then show the general case.

1 Coinductive predicates on automata

Let $\langle \epsilon, \delta \rangle \colon X \to 2 \times X^A$ be a deterministic automaton. Recall that the language semantics is a function $\mathsf{beh} \colon X \to \mathcal{P}(A^*)$, mapping each state to the language that it accepts. We say two states x, y are *language equivalent* if $\mathsf{beh}(x) = \mathsf{beh}(y)$.

We denote by Rel_X the set of all relations $R \subseteq X \times X$ on the state space of our automaton. This set is ordered by inclusion. Consider the function $b \colon \mathsf{Rel}_X \to \mathsf{Rel}_X$, defined by

$$b(R) = \{(x, y) \mid \epsilon(x) = \epsilon(y) \text{ and for all } a \in A. \, , (\delta(x)(a), \delta(y)(a)) \in R\} \, . \quad (1)$$

We are particularly interested in relations R such that $R \subseteq b(R)$; such a relation is a *bisimulation*. Indeed, we have $R \subseteq b(R)$ iff for all $(x, y) \in R$:

1. $\epsilon(x) = \epsilon(y)$, and
2. for all $a \in A$: $(\delta(x)(a), \delta(y)(a)) \in R$.

That looks familiar: it is the concrete notion of bisimulation between automata that we've seen before in this course.

The union of a set of bisimulations is again a bisimulation. In particular, we let $\sim$ be the union of *all* bisimulations. This is then the *greatest* bisimulation with respect to set inclusion. Since it is the greatest we immediately obtain the following principle for all $x, y \in X$ and $R \in \mathsf{Rel}_X$:

$$\frac{(x, y) \in R \subseteq b(R)}{x \sim y}$$

Why is this interesting? The point is that the greatest bisimulation characterises language equivalence.

Proposition 1.1. *For all* $(x, y) \in X$*:* $x \sim y$ *iff* $\mathsf{beh}(x) = \mathsf{beh}(y)$.

We've already seen a proof of this: in one of the previous lectures, we proved that beh is the unique coalgebra homomorphism to the final coalgebra, and that (hence) bisimilarity coincides with language equivalence. All in all, we obtain the following proof principle for language equivalence:

$$\frac{(x, y) \in R \subseteq b(R)}{\mathsf{beh}(x) = \mathsf{beh}(y)}$$

which we refer to as a *coinductive proof principle*. So far, so good: this is just a slight rephrasing of things that we already got from the coalgebraic perspective on automata.

But we can also do something slightly different: consider $b' \colon \mathsf{Rel}_X \to \mathsf{Rel}_X$ defined by

$$b'(R) = \{(x, y) \mid \epsilon(x) \leq \epsilon(y) \text{ and for all } a \in A.\ , (\delta(x)(a), \delta(y)(a)) \in R\}\,. \quad (2)$$

A relation R satisfies $R \subseteq b'(R)$ if and only if for all $(x, y) \in R$:

1. $\epsilon(x) \leq \epsilon(y)$, and
2. for all $a \in A$: $(\delta(x)(a), \delta(y)(a)) \in R$.

We call such a relation a *simulation*. This is different than bisimulations: here, we only require that whenever x is accepting, y is accepting as well, but when x is not accepting we don't care whether y accepts or not. What does this capture?

To see this, we first observe that, similar to the case of bisimulations, the union of an arbitrary set of simulations is again a simulation. Let's write $\preceq$ for the union of all simulations. What is this?

Proposition 1.2. *For all* $(x, y) \in X$*:* $x \preceq y$ *iff* $\mathsf{beh}(x) \subseteq \mathsf{beh}(y)$.

It captures language inclusion. Thus, we've captured language inclusion as the greatest simulation. Hence, we obtain the following proof principle for language inclusion.

$$\frac{(x, y) \in R \subseteq b'(R)}{\mathsf{beh}(x) \subseteq \mathsf{beh}(y)}$$

Great; so to prove that the language of a state x is included in the language of a state y, it suffices to come up with a simulation that contains (x, y).

Language inclusion is not quite behavioural equivalence, and it is not immediately clear how it would follow from the coalgebraic perspective on automata. Nevertheless, we would like to view the above proof principle as *coinductive*: with the rather unprecise idea that we capture it as the "greatest" relation satisfying a few properties, just like with bisimulations. In order to make this more precise, we use lattices.

2 Lattices

A *partially ordered set*, or *poset* for short, is a pair $(P, \leq)$ where $\leq \subseteq P \times P$ is reflexive, transitive and anti-symmetric, that is, for all $x, y, z \in P$:

1. $x \leq x$,
2. $x \leq y$ and $y \leq z$ imply $x \leq z$,
3. $x \leq y$ and $y \leq x$ imply $x = y$.

The appropriate notion of morphisms between lattices is given by the notion of monotone function. Given posets $(P, \leq_P)$ and $(Q, \leq_Q)$, a function $f \colon P \to Q$ is *monotone* if for all $x, y \in P$: $x \leq_P y$ implies $f(x) \leq_Q f(y)$.

Let $(P, \leq)$ be a poset. An element $x \in P$ is called *top* if for all $y \in P$: $y \leq x$. An element $x \in P$ is called *bottom* if for all $y \in P$: $x \leq y$. A top element does not need to exist, but if it does, it is unique. In that case we denote it by $\top$. Similarly for a bottom element, which we denote by $\bot$.

An *upper bound* of a set $S \subseteq P$ is an element $x \in P$ such that for all $y \in S$: $y \leq x$. The *least upper bound* or *supremum* or *join* of a set S is an element $\bigvee S$ such that

- $\bigvee S$ is an upper bound of S, and
- if x is an upper bound of S then, $\bigvee S \leq x$.

The dual notion is that of a *greatest lower bound*, also called *infimum* or *meet*. Given elements $x, y \in P$, we write $x \vee y$ for $\bigvee\{x, y\}$ and $x \wedge y$ for $\bigwedge\{x, y\}$.

A *lattice* is a poset P which has a bottom and a top element, and such that $x \vee y$ and $x \wedge y$ exist for all $x, y \in P$. A *complete lattice* is a poset P in which the join $\bigvee S$ and the meet $\bigwedge S$ exist for every subset $S \subseteq P$.

Example 2.1. There are many, many examples of posets, (complete) lattices. We only list a few that we need in the examples in the next sections.

1. Given a set X, the poset $(\mathsf{Rel}_X, \subseteq)$ of relations on X ordered by inclusion is a complete lattice. The join of a set of relations is given by union, and the meet by intersection; the top element is $X \times X$, and the bottom element is the empty relation on X.
2. A very similar example is the complete lattice of *predicates* on X: this is just the set of subsets $P \subseteq X$ of X. Again the order is set inclusion. We denote this complete lattice by Pred_X.
3. How about the set of natural numbers ordered as usual? Is it a lattice? Is it complete? How about the set of integers? (These are exercises!)

3 Coinduction in a lattice

Now that we know what a lattice is, let's focus on coinduction again. The example in the first section is the lattice Rel_X of relations; we looked at a function $b\colon \mathsf{Rel}_X \to \mathsf{Rel}_X$ on it, and in particular the relations R such that $R \subseteq b(R)$.

More generally, let $b\colon P \to P$ be a monotone function on a poset. A *post fixed-point* of b is an element $x \in P$ such that $x \leq b(x)$. A *fixed point* of b is an element such that $b(x) = x$. A function b may or may not have a *greatest fixed point*, written $\mathsf{gfp}(b)$, which is a fixed point of b such that for every fixed point x of b: $x \leq \mathsf{gfp}(b)$. In a complete lattice, such a greatest fixed point always exists; this is a fundamental result known as the Knaster-Tarski theorem.

Theorem 3.1. *Let $b\colon P \to P$ be a monotone function on a complete lattice P. Then b has a greatest fixed point $\mathsf{gfp}(b)$, given by*

$$\mathsf{gfp}(b) = \bigvee \{x \in P \mid x \leq b(x)\}\,.$$

Proof. Let $z = \bigvee\{x \in P \mid x \leq b(X)\}$. We should show that z is a fixed point: $z = b(z)$, and that it is the greatest fixed point. We leave this as an exercise. Hint: to prove that z is a fixed point, first prove that $z \leq b(z)$, then that $b(z) \leq z$. □

With this theorem in hand, we have a basic infrastructure to speak about coinduction, starting from a monotone function $b\colon P \to P$ on a complete lattice. The above theorem tells us that the greatest fixed point of b exists, and it gives us the following principle:

$$\frac{x \leq b(x)}{x \leq \mathsf{gfp}(b)}$$

which we may refer to as a *coinductive proof principle*.

Let's look at the first example of the first section. There, the lattice of interest is Rel_X, the relations on the state space X of an automaton. The function $b\colon \mathsf{Rel}_X \to \mathsf{Rel}_X$ given in (1) is the one we're interested in; a post-fixed of b is a relation R such that $R \subseteq b(R)$, which we recognise as a bisimulation. This function is monotone (exercise) and hence, by the above theorem, it has a greatest fixed point $\mathsf{gfp}(b)$, given as the join of all post-fixed points. But this is just the greatest bisimulation that we've seen before! And the coinductive proof principle tells us that any relation R is contained in the greatest one. This is useful, as we proved (separately) that the greatest bisimulation coincides with language equivalence. A similar story holds for simulation, where we use b' as defined in (2).

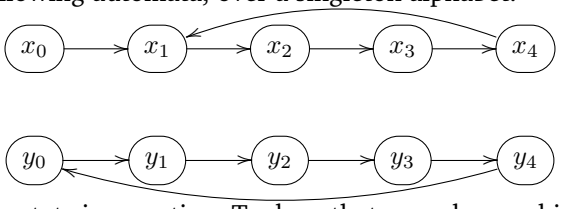
This scheme is pretty general: it allows us to define many kinds of predicates on many kinds of systems. As another example, let $\langle o, f\rangle\colon X \to \mathbb{N} \times X$ be a stream system. We look at the complete lattice Pred_X of predicates (subsets) on X, and define $b\colon \mathsf{Pred}_X \to \mathsf{Pred}_X$ by

$$b(P) = \{x \in X \mid o(x) \leq f(o(x)) \text{ and } f(x) \in P\}\,.$$

What is a post-fixed point of b? What is the greatest fixed point?

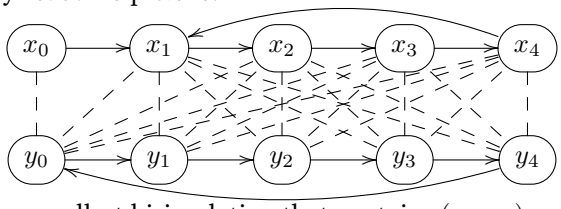
4 Bisimulation up-to: automata

Consider the following automata, over a singleton alphabet.



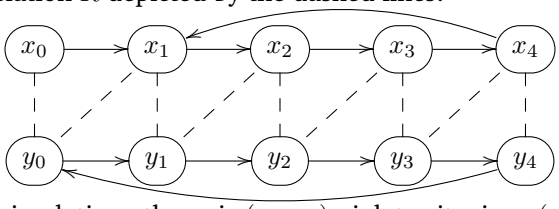
For simplicity, no state is accepting. To show that x_0 and y_0 are bisimilar, we may show that they are contained in a bisimulation, that is, $(x_0, y_0) \in R$ for some R such that $R \subseteq b(R)$, with b as defined in (1). The typical approach is to start with $\{(x_0, y_0)\}$; since that is not a bisimulation, we expand to $\{(x_0, y_0), (x_1, y_1)\}$; again not a bisimulation, and so on. This will go on for a while!

If we depict the bisimulation that we obtain in this way with dashed arrows, we obtain a very readable picture.



This is in fact the smallest bisimulation that contains (x_0, y_0).

Luckily we can do better, by using an up-to technique. We've seen up-to techniques before in the context of bisimulations between languages. Consider the following relation R depicted by the dashed lines:



This is not a bisimulation: the pair (x_4, y_3) violates it, since (x_1, y_4) is not related. However, (x_1, y_4) is related by the least equivalence relation $\mathsf{eqv}(R)$ containing R. More precisely, $\mathsf{eqv}(R)$ is the least relation satisfying the following rules:

$$\frac{x\ R\ y}{x\ \mathsf{eqv}(R)\ y} \qquad \frac{}{x\ \mathsf{eqv}(R)\ x} \qquad \frac{x\ \mathsf{eqv}(R)\ y}{y\ \mathsf{eqv}(R)\ x} \qquad \frac{x\ \mathsf{eqv}(R)\ y \quad y\ \mathsf{eqv}(R)\ z}{x\ \mathsf{eqv}(R)\ z}$$

With that in place, for the above relation given by the dashed lines, we have $R \subseteq b(\mathsf{eqv}(R))$. Such a relation is called a *bisimulation up to equivalence*.

And now what? Well, it turns out that bisimulations up to equivalence suffice to prove language equivalence. This is called *soundness*, and we may express it nicely as another coinduction principle:

$$\frac{(x, y) \in R \subseteq b(\mathsf{eqv}(R))}{\mathsf{beh}(x) = \mathsf{beh}(y)}$$

So bisimulations up to equivalence suffice to prove language equivalence. This underlies a classical algorithm by Hopcroft and Karp, for language equivalence of deterministic automata. It starts by relating the two states that are to be compared, and keeps adding pairs until the result is a bisimulation up to equivalence (or a counterexample is found).

4.1 Non-deterministic automata

In the previous lecture, we've talked a lot about determinisation. Given a non-deterministic automaton

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

we transform it into a deterministic automaton

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

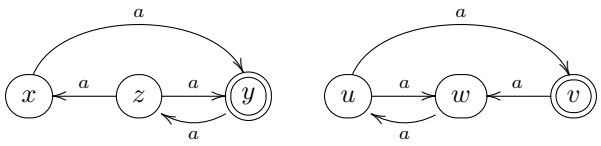
using the powerset construction. The familiar coalgebra homomorphism

$$\mathsf{beh} \colon \mathcal{P}(X) \to \mathcal{P}(A^*)$$

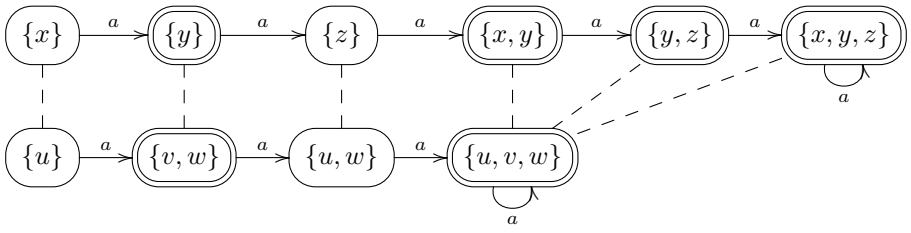
gives us the language semantics. More precisely, the language of a state $x \in X$ is given by $\mathsf{beh}(\{x\})$.

The first observation is that, to check whether two states $x, y \in X$ of the non-deterministic automaton accept the same languages, it suffices to construct a bisimulation $R \subseteq \mathcal{P}(X) \times \mathcal{P}(X)$ on the deterministic automaton $\langle \epsilon^\sharp, \delta^\sharp \rangle$ that contains the pair $(\{x\}, \{y\})$. Or we compute a bisimulation up to equivalence that contains it; that also suffices.

But it turns out that, for non-deterministic automata, we can do something even smarter to compare their languages—this idea, and examples below, are from [1]. Consider the following example. For simplicity, we view the automaton below as a single automaton; and we're interested in comparing the language of x and u.



If we determinise these automata, we get (as a part):



The dashed lines give a bisimulation between $\{x\}$ and $\{u\}$. That suffices to prove language equivalence. But we can do better. Consider

$$R = \{(\{x\}, \{u\}), (\{y\}, \{v, w\}), (\{z\}, \{u, w\})\},$$

that is, the first three dashed lines from the left. It is not a bisimulation, since $(\{x, y\}, \{u, v, w\})$ is not in R. However, the missing pair is a (pointwise) *union* of pairs that are already in R. This shows that R is a *bisimulation up to congruence*.

To define that notion in general, for a relation $R \subseteq \mathcal{P}(X) \times \mathcal{P}(X)$, we define the *congruence closure* $\mathsf{cgr}(R)$ as the least relation such that

$$\frac{X\ R\ Y}{X\ \mathsf{cgr}(R)\ Y} \qquad \frac{}{X\ \mathsf{cgr}(R)\ X} \qquad \frac{X\ \mathsf{cgr}(R)\ Y}{Y\ \mathsf{cgr}(R)\ X} \qquad \frac{X\ \mathsf{cgr}(R)\ Y \quad Y\ \mathsf{cgr}(R)\ Z}{X\ \mathsf{cgr}(R)\ Z}$$

$$\frac{X_1\ \mathsf{cgr}(R)\ Y_1 \quad X_2\ \mathsf{cgr}(R)\ Y_2}{(X_1 \cup X_2)\ \mathsf{cgr}(R)\ (Y_1 \cup Y_2)}$$

A *bisimulation up to congruence* is a relation $R \subseteq \mathcal{P}(X) \times \mathcal{P}(X)$ such that $R \subseteq b(\mathsf{cgr}(R))$, where b is the function we've talked about all the time (Equation (1)). The good news is that this is a sound technique.

Theorem 4.1. *Let*

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

be the determinisation of a deterministic automaton $(X, \langle \epsilon, \delta \rangle)$*. If* $R \subseteq \mathcal{P}(X) \times \mathcal{P}(X)$ *is a bisimulation up to congruence, then* $\mathsf{cgr}(R)$ *is a bisimulation.*

As a consequence, we get the following principle for proving language equivalence.

$$\frac{(\{x\}, \{y\}) \in R \subseteq b(\mathsf{cgr}(R))}{\mathsf{beh}(x) = \mathsf{beh}(y)}$$

So to prove that states x, y are language equivalent, it suffice to show that the singletons $\{x\}$ and $\{y\}$ are related by a bisimulation up to congruence.

5 A theory of up-to techniques

In the previous sections, we've seen a few nice proof techniques: bisimulations up-to equivalence, and bisimulations up to congruence. Also, in one of the previous lectures, we've seen a different proof technique, also called bisimulation up to congruence, but where the congruence closure was a bit different: there, we looked at the regular operations (union, concatenation, Kleene star) on languages, rather than the structure of determinised automata.

Now one may wonder: how to actually prove the soundness of these techniques? Can we say something more general about it? Here, the notion of coinduction in a complete lattice helps a lot.

So let $b \colon P \to P$ be a monotone function on a complete lattice. What is an up-to technique? We start with some monotone function $f \colon P \to P$. Before,

we've looked at post-fixed points of the form $x \leq b(x)$; now, we look at post-fixed points $x \leq b(f(x))$. And we say f is b-*sound* if the following principle holds:

$$\frac{x \leq b(f(x))}{x \leq \mathsf{gfp}(b)}$$

In the lattice of relations, this means that if $R \subseteq b(f(R))$, R should be contained in the greatest fixed point of b.

That gives at least the soundness, but how do we prove it? Well, here we observe that the various functions decompose into simpler ones: the congruence closure is built from the equivalence closure and something called the contextual closure, and the equivalence closure in turn decomposes as transitive, symmetric, reflexive closure. So we'd like to prove soundness in terms of these simpler functions.

The problem is that this doesn't work, in general: the composition of sound functions is not necessarily sound. The solution is to look at a slightly stronger notion: that of a *compatible function*. With f,b as above, we say f is b-compatible if $f(b(x)) \leq b(f(x))$ for all $x \in P$. This has two important properties:

- if f is b-compatible, then f is b-sound;
- if f, g are both b-compatible, then $f \circ g$ is b-compatible.

There are many more ways of combining compatible functions; but this is the most crucial one. If you'd like to read more about this, the paper [1] is a good start, as it explains these things in the basic context of automata. Up-to techniques, especially in the context of coalgebras, are also the main topic of my PhD thesis.

References

[1] Filippo Bonchi and Damien Pous. Checking NFA equivalence with bisimulations up to congruence. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2013, proceedings*, pages 457–468. ACM, 2013. 6, 8