

Logical specification and coinduction

Robert J. Simmons
rjsimmon@cs.cmu.edu

November 29, 2008

This note primarily follows Nielson et al's in [5], which cites [4, Chapter 4] as one of the few other decent introductions to coinduction. I haven't gotten my hands on the latter reference yet.

1 Specification

Nielson et al. [5] use a coinductive specification to describe *acceptability* of a static analysis (notably, OCFA). We will start with a simpler coinductive specification; the property in English that we want to talk about is “A finite database G is an acceptable analysis of x if $\text{at}(y) \in G$ for all y less than x .”

We will start using a similar notation to Nielson: $G \models x$, where G is a (finite) database of facts and e is a term, means that G is an acceptable analysis of x .

1.1 Inductive specification

In the first example, we will give a specification that makes sense as an inductive specification by using unary natural numbers $n ::= z \mid s(n)$. Then, we can define $G \models n$ by structural induction on n .

- $G \models z$ always.
- $G \models s(n)$ iff $\text{at}(n) \in G$ and $G \models n$.

The expression n always gets smaller in places where the specification refers to itself, so for any n , we could just unravel the entire specification to a finite specification that isn't self-referential:

- $G \models s(s(s(z)))$ iff $\text{at}(s(s(z))) \in G$ and $\text{at}(s(z)) \in G$ and $\text{at}(z) \in G$.

This specification will consider acceptable any superset of the set $\{\text{at}(s(s(z))), \text{at}(s(z)), \text{at}(z)\}$ – a set with “junk” in it is acceptable, but the set must include at least those elements – this is a property of acceptability relations in general.

1.2 Coinductive specification

In this example, we will use not unary natural numbers but arbitrary constants a, b, c, \dots – the notion of “less than” will be defined by a set of facts $\text{succ}(x, y)$ in the database. The specification looks like this:

- $G \models a$ iff $\forall b$ such that $\text{succ}(b, a) \in G$, $\text{at}(b) \in G$ and $G \models b$.

If the succ relation for a particular G forms a partial order, then it doesn't matter whether the specification is understood inductively or coinductively – the action of running the successor relation backwards is a well-founded induction.

A problem arises as soon as we consider a set where the successors form a cycle, such as the set $\{\text{succ}(a, b), \text{succ}(b, a), \text{at}(a), \text{at}(b)\}$, which has a two-element cycle. Intuitively, this meets our informal specification – the set is an acceptable approximation for both a and b because it contains $\text{at}(x)$ for all x that can be reached by running the successor relation backwards, even if “less than” is no longer an especially meaningful property.

Our previous notion of “unfolding” doesn't work because it never ends – the unfolding turns into the following, where we cannot eliminate the $G \models a$ on the right:

- $G \models a$ iff $\text{at}(b) \in G$ and $\text{at}(a) \in G$ and $G \models a$.

If we treat our definition as inductive, this means that the set $\{\text{succ}(a, b), \text{succ}(b, a), \text{at}(a), \text{at}(b)\}$ is not an acceptable approximation – no approximations with cycles are, in fact. If we treat our definition as coinductive, however, we essentially assume $G \models a$ is satisfied and so must only show $\text{at}(b) \in G$ and $\text{at}(a) \in G$, which is true for the set we were considering.

2 Logic programming

In this entire development, we have treated the database G as a finite set of ground facts, and the database G is constant throughout the entire analysis. In other words, the specifications we have considered bear a resemblance to logic programs, where the facts queried from the database are ground atomic facts (with positive polarity).

We will attempt to gain some insight and leverage on our specifications using logic programming. If we interpret the the if and only if relationship in a specification as “if,” the result suggests a top-down logic program for deciding $G \models x$ for a particular G and x . If, on the other hand, we interpret the if and only if relationship in a specification as “only if,” the result suggests a bottom-up logic program for computing a G for a given x such that $G \models x$.

There are challenges with both of these approaches, but we will suggest that, while the problems of the first, top-down, approach are significant and the approach seems to be limited, the bottom-up approach appears to be a powerful way to connect the kind of bottom-up analysis presented in [7] with the kinds of coinductive specification presented by Nielson et al. in [5].

2.1 Top-down logic programming

If we turn $G \models n$ into $\text{eval}(n)$, leaving G implicit, then we can rewrite the two rules in the inductive specification as $\text{eval}(z) \Leftrightarrow \top$, and $\forall n. \text{eval}(s(n)) \Leftrightarrow \text{at}(n) \wedge \text{eval}(n)$.

The notion here is to use logic programming to capture the “if” direction, and then use the concept of “iff completion” to capture the “only if” direction of the specification [6, Section 8.7]. The result will be a logic program that, if run with additional axioms for the elements of G , will act as a *decision procedure* for acceptability. In particular, we will have $G \models n$ iff $\text{eval}(n)$ can be established by top-down logic search.

- $\text{eval}(z) \text{ :- true.}$

- $\text{eval}(s(N)) \text{ :- } \text{at}(N), \text{eval}(N).$

This doesn't work at all for our example of coinductive specification. It would translate to $\forall a.\text{eval}(a) \Leftrightarrow \forall b.\text{succ}(b, a) \Rightarrow \text{at}(b) \wedge \text{eval}(b)$. The "if" direction could be written in an odd hybrid of Twelf and Prolog syntax:

- $\text{eval}(A) \text{ :- } \{B\} \text{succ}(B, A) \text{ -> } \text{at}(B), \text{eval}(B).$

The arrow -> above captures the wrong function space, however. The uniform proof semantics of logic programs [3] translates the single subgoal as "introduce a new constant B and a new axiom $\text{succ}(B, A)$ and try to prove $\text{at}(B)$ and $\text{eval}(B)$." What we want is more of what Licata et al. would call the "computational arrow" [2], because the intended meaning is "for each constant B such that $\text{succ}(B, A)$ is already true, prove $\text{at}(B)$ and $\text{eval}(B)$."¹

Even if we could endow top-down logic programming with the correct computational behavior, we would still have a looping issue in cases where the successor relation is not a partial order. Generalizations of top-down logic programming aimed at prevent looping, such as tabling, would not help, because the problem is still one of encountering a subgoal $\text{eval}(a)$ while trying to prove $\text{eval}(a)$.

Going back to iff completion, a possibly nonterminating logic program has three possible outcomes: success, failure, or nontermination. If the semantics of the logic program match up with the specification, it might be the case that success corresponded to matching the inductive specification, whereas nontermination corresponded to matching the coinductive specification. There are more fundamental problems, however; we will not consider this particular thread further here.

In conclusion, interpreting the \Leftrightarrow as \Leftarrow suggests reading acceptability specifications as top-down logic programs for deciding acceptability of an analysis, but this approach suffers from a mismatch with the semantics of uniform proofs for even inductive interpretations and unresolved termination issues when dealing with coinductive interpretations.

2.2 Bottom-up logic programming

If we return to the logical rewriting of the specification $\text{eval}(z) \Leftrightarrow \top$, and $\forall n.\text{eval}(s(n)) \Leftrightarrow \text{at}(n) \wedge \text{eval}(n)$, but instead consider the "only if" direction, we are given the following program, which we read as a saturating logic program.

$$\frac{\text{eval}(z)}{\text{eval}(s(n))} \quad \frac{\text{at}(n)}{\text{eval}(n)}$$

What to do with these rules? If we want to compute an acceptable analysis G for a given n , we would hope to be able to add $\text{eval}(n)$ to a database and run to saturation under those rules, and have the saturated database G be an acceptable analysis.

The same process works for the second example. The formula $\forall a.\text{eval}(a) \Rightarrow \forall b.\text{succ}(b, a) \Rightarrow \text{at}(b) \wedge \text{eval}(b)$ is equivalent to $\forall a, b.\text{eval}(a) \wedge \text{succ}(b, a) \Rightarrow \text{at}(b) \wedge \text{eval}(b)$, which we write as

$$\frac{\text{eval}(a), \text{succ}(b, a)}{\text{at}(b), \text{eval}(b)}$$

¹Check- does Miller's definitional reflection address this problem?

This will work just as well in the case where the pre-defined successor relation is a partial order or not.

3 Thoughts and questions

3.1 Proofs and propositions

- The specific example (less-than) satisfies the particular coinductive specification.
- What is the fragment for which a straightforward rewriting works? This isn't like iff-completion, because it's certainly not the case that if we have $\text{at}(b)$, $\text{eval}(b)$, and $\text{succ}(b, a)$ then we have $\text{eval}(a)$ – the saturating analysis only goes in one direction. Why, in general, does the completed database satisfy the coinductive specification that it describes?
- It is my expectation that this process uncovers not only *some* acceptable analysis but the *least* acceptable analysis. Nielson et al. points out that the existence of least solutions is not ensured, so what are the criteria that are necessary to find it? How do we show that it establishes the least acceptable analysis in this straightforward manner, which means we get to skip three steps that Nielson has to go through.
- What else is there to prove/show/consider?

3.2 Compilation

The difference between the straightforward translation and the analysis presented in [7] is that conclusions that are functions are captured by `comp` propositions. For instance, the literal translation of the clause for function application in [5] is this (to invent a bit of notation):

$$\frac{\text{eval}((t_1^{l_1} \ t_2^{l_2})^l)}{\text{eval}(t_1^{l_1}) \quad \text{eval}(t_2^{l_2}) \quad \forall x, t_0, l_0. \text{return}(l_1, \lambda(x, t_0^{l_0})) \rightarrow \text{eval}(t_0^{l_0}) \wedge (\forall v_2. \text{return}(l_2, v_2) \rightarrow \text{bind}(x, v_2)) \wedge (\forall v_0. \text{return}(l_0, v_0) \rightarrow \text{return}(l, v_0))}$$

The above is closer to the presentation of substructural operational semantics in [1], but if we wanted to have a comparable notation that is closer to what is presented in [7], it would look something like this:

$$\frac{\text{eval}((t_1^{l_1} \ t_2^{l_2})^l)}{\text{eval}(t_1^{l_1}) \quad \text{eval}(t_2^{l_2}) \quad \text{comp}(l_1, l_2, l)} \quad \frac{\text{comp}(l_1, l_2, l) \quad \text{return}(l_1, \lambda(x, t_0^{l_0}))}{\text{eval}(t_0^{l_0}) \quad \text{comp}(l_2, x)} \quad \frac{\text{comp}(l_2, x) \quad \text{return}(l_2, v_2)}{\text{bind}(x, v_2)} \quad \frac{\text{comp}(l_0, l) \quad \text{return}(l_0, v_0)}{\text{return}(l, v_0)}$$

What's the simple, clean, and general way of showing a correspondence between these two ways of doing things?

References

- [1] I. Cervesato, F. Pfenning, D. Walker, and K. Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, School of Computer Science, Carnegie Mellon University, Mar. 2002. Revised May 2003.
- [2] D. R. Licata, N. Zeilberger, and R. Harper. Focusing on binding and computation. In *LICS '08: Proceedings of the 2008 23rd Annual IEEE Symposium on Logic in Computer Science*, pages 241–252, Washington, DC, USA, 2008. IEEE Computer Society.
- [3] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Ann. Pure Appl. Logic*, 51(1-2):125–157, 1991.
- [4] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
- [5] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.
- [6] F. Pfenning. Logic Programming. Course notes, 15-815K, Fall 2006.
- [7] R. J. Simmons and F. Pfenning. Linear Logical Approximations. In *PEPM '09: Proceedings of the 2009 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, New York, NY, USA, 2009. ACM. To appear.