Logical specification and coinduction

Robert J. Simmons

Carnegie Mellon University, Pittsburgh PA 15213, USA,
rjsimmon@cs.cmu.edu,
WWW home page: http://www.cs.cmu.edu/∼rjsimmon

Abstract. The notion of an "acceptable analysis" is an important concept in program analysis. Acceptability of a model is defined by coinduction, that is, as a greatest fixed–point. We describe a class of specifications that includes our examples of acceptable analyses, and then we interpret that specification as a forward reasoning logic program. If this forward reasoning logic program reaches the point of saturation (where no new facts are derivable), then the completed database satisfies the coinductive specification.

Control flow analyses of functional programs (such as 0CFA) are generally specified by means of a coinductive specification of what it means to be an *acceptable control flow analysis*, as described in Nielson et al. [2]. However, this specification is usually not thought to provide much insight into the design of an analysis satisfying the specification. For instance, Nielson et al. first define acceptability for 0CFA and then, later, describe a less precise analysis that may be run by inducting over the structure of the program expression. Similarly, Van Horn and Mairson describe *instrumented evaluators* which provide a procedural implementation of the declarative specification [5, 7, 6]. We will address this troubling disconnect between specification and implementation using forward–chaining logic programming.

McAllester was instrumental in showing that static analyses could be concisely represented and efficiently run as logic programs [1], and previous work by Simmons and Pfenning has shown that sound–by–construction program analyses in the style of McAllester can be derived from a logical presentation of a programming language's dynamic semantics [4]. One of the analyses described in that paper is a 0CFA–like program approximation derived from dynamic semantics of the call–by–value lambda calculus, but this approximation was never related to a coinductive specification of 0CFA.

In this paper, we provide the missing link, a formal connection between coinductive specifications and logic programs that compute minimal models satisfying those coinductive specifications. In Section 1 we describe a language for describing sets of acceptable databases, including (co)inductively defined sets of databases, and give examples. In Section 2 we show how to take a coinductive specification and derive a forward–chaining logic program that computes databases satisfying the coinductive specification. Finally, in Section 3, we give the specification of an acceptable 0CFA analysis along with its corresponding logic program and discuss the relationship with the derived analysis in [4].

1 Specifying properties of databases

Consider a database holding ground atomic propositions of the form $\mathsf{reachable}(n)$ and $\mathsf{succ}(n, n')$. We want to specify those databases G where, for a given number n, $\mathsf{reachable}(n')$ holds for all (eventual) successors of n as defined by the succ relation. This specification, written in the style of Nielson et al., might look like this:

$$G \models n \text{ iff for all } n' \text{ where } \mathsf{succ}(n, n') \in G,\ \mathsf{reachable}(n') \in G \text{ and } G \models n' \quad (1)$$

In other words, $G \models n$ (which we will refer to as an *acceptability relation*) is defined as a fixed–point. If we consider a database G that contains $\mathsf{succ}(\mathsf{n_0}, \mathsf{n_{1A}})$ and $\mathsf{succ}(\mathsf{n_0}, \mathsf{n_{1B}})$ (and no other succ propositions), then $G \models \mathsf{n_0}$ precisely if G also contains $\mathsf{reachable}(\mathsf{n_{1A}})$ and $\mathsf{reachable}(\mathsf{n_{1B}})$.

Now consider a database G' that contains $\mathsf{succ}(\mathsf{n_0}, \mathsf{n_1})$ and $\mathsf{succ}(\mathsf{n_1}, \mathsf{n_0})$ (and no other succ propositions). Under what additional conditions does $G' \models \mathsf{n_0}$? It depends! If the acceptability relation is defined coinductively (that is, as the greatest relation that meets the specification), then the answer is that as long as G' also contains $\mathsf{reachable}(\mathsf{n_0})$ and $\mathsf{reachable}(\mathsf{n_1})$, we have that $G' \models \mathsf{n_0}$ holds. However, if the acceptability relation is defined inductively (as the least relation that meets the specification), then there is *no* database G' containing $\mathsf{succ}(\mathsf{n_0}, \mathsf{n_1})$ and $\mathsf{succ}(\mathsf{n_1}, \mathsf{n_0})$ such that $G' \models \mathsf{n_0}$. The culprit is the cycle $\mathsf{n_0} \to \mathsf{n_1} \to \mathsf{n_0}$ in the successor relation; it means that to prove $G' \models \mathsf{n_0}$ we must prove $G' \models \mathsf{n_1}$, which means we must prove $G' \models \mathsf{n_0}$, and so on forever. If we hope to prove that for any G and n there exists some $G' \supseteq G$ such that $G' \models n$, we must therefore consider the coinductive interpretation of the acceptability relation.

1.1 A language of database specifications

We will develop what at first can be seen as merely a more concise language for specifying properties of databases, where databases are define as possibly infinite sets of ground facts $\mathsf{a}(\bar{t})$.

Consider the grammar below, where P is a proposition describing a set of databases. Q is an atomic proposition $\mathsf{a}(\bar{t})$ where the terms $\bar{t}$ may include free variables.

$$P ::= Q \mid P \wedge P \mid \forall x.P \mid Q \to P \mid t_1 \doteq t_2 \to P \quad (2)$$

We assume that we have some fixed signature Σ that allows us to define the largest possible database as a Herbrand universe. The set of all possible databases is the power set of this database, and we give an interpretation of a proposition P (which we write as $|P|$) as a set of databases, that is, as a member of the power set of the set of all possible databases. We will also write $D \models P$ to mean that $D \in |P|$.

We define what it means to be a member of $|P|$ inductively on the structure of P. The definition has a "classical" feel (in the sense of classical logic), due to the way $Q \to P$ and $t_1 \doteq t_2 \to P$ are defined.

$D \models Q$ iff $Q \in D$.
$D \models P_1 \wedge P_2$ iff $D \models P_1$ and $D \models P_2$.
$D \models \forall x.P$ iff for all ground terms t, $D \models P[t/x]$.
$D \models Q \to P$ iff either $Q \notin D$ or $D \models P$.
$D \models t_1 \doteq t_2 \to P$ iff either $t_1 \neq t_2$ or $D \models P$.

Examples This language does not yet give us the ability to specify acceptable databases as described by Eqn. 1, but many other sets of databases can be characterized; we will give three examples.

Every database containing at least $\mathsf{reachable}(\mathsf{n_0})$ and $\mathsf{reachable}(\mathsf{n_1})$ can be specified by either the proposition $\mathsf{reachable}(\mathsf{n_0}) \wedge \mathsf{reachable}(\mathsf{n_1})$ or, equivalently, by the proposition $\forall n.\ (n \doteq \mathsf{n_0} \to \mathsf{reachable}(n)) \wedge (n \doteq \mathsf{n_1} \to \mathsf{reachable}(n))$.

If we want to consider the transitive closure of the succ relation from before (call it succ^*), then we can specify the analogue of Eqn. 1 as follows:

$$\lambda n. \forall n'.\ \mathsf{succ}^*(n, n') \to \mathsf{reachable}(n') \tag{3}$$

Notice the λn, indicating that Eqn. 3 is actually a function taking a term n and returning a proposition.

Finally, we can specify the databases containing $\mathsf{nat}(n)$ for every unary natural number n as follows:

$$\mathsf{nat}(\mathsf{z}) \wedge \forall n.\ \mathsf{nat}(n) \to \mathsf{nat}(\mathsf{s}(n)) \tag{4}$$

This allows us to point out an important distinction between the *database itself* and *sets of databases*. Depending on the signature Σ, there may be zero, one, or an infinite number of databases G satisfying the specification in Eqn. 4. However, each database G within this set must be countably infinite, as it has to include $\mathsf{nat}(n)$ for every unary natural number n in order to meet the specification.

1.2 Greatest fixed–points

We say that $P_1 \Rightarrow P_2$ if $D \models P_1$ implies $D \models P_2$ (in other words, if $|P_1| \subseteq |P_2|$). We can lift this notion to propositions with free variables: $P_1' \Rightarrow P_2'$ if, for all $\bar{t}$, $P_1'(\bar{t}) \Rightarrow P_2'(\bar{t})$.

We are interested in the *monotone* functions F from propositions (with free variables) to propositions (with the same free variables). Monotonicity of F means that for all ground terms $\bar{t}$, and propositions P_1, and P_2, if $P_1(\bar{t}) \Rightarrow P_2(\bar{t})$ then $(FP_1)(\bar{t}) \Rightarrow (FP_2)(\bar{t})$ holds.

Lemma 1. *If F is a* substitution function $\lambda P.\lambda \bar{x}.P_f(\bar{x})$*, that is, if there is some proposition P_f with one or more "holes" and FP simply plugs P into all of the holes, then F is monotone.*

Proof. By induction on the structure of P_f, the proposition with holes.

If F is monotone, then by the Knaster–Tarski theorem there exists a greatest (and least) fixed–point of F. Using greatest fixed–points νF, we can finally give a specification corresponding to Eqn. 1:

$$\nu\lambda P.\ \lambda n.\ \forall n'.\ \mathsf{succ}(n, n') \to \mathsf{reachable}(n') \wedge P(n) \tag{5}$$

2 Implementing specifications as logic programs

We have described a clean, logical notation for specifying sets of databases, including coinductively defined sets of databases; now we want to use logic programming to compute databases which satisfy those specifications. Our specification language is already a subset of a logic programming language [3]. This observation allows us to state our main result:

Theorem 1. *If F is a substitution function that does not include the predicate symbol* eval*, then for all terms $\bar{t}$, if* $\mathsf{eval}(\bar{t}) \wedge \forall \bar{y}.\ \mathsf{eval}(\bar{y}) \rightarrow (F(\mathsf{eval}))(\bar{y})$ *is a range–restricted logic program that runs to saturation from an initial database G to produce a database G', then $G' \models (\nu F)(\bar{t})$.*

Proof. By Lemma 2, we know that $\mathsf{eval}(\bar{t}) \wedge \forall \bar{y}.\ \mathsf{eval}(\bar{y}) \rightarrow (F(\mathsf{eval}))(\bar{y})$ is a pre–fixed–point of F. By premise, we can produce a saturated database G' by exhaustive forward reasoning, which by Lemma 3 means that we have $G' \models \mathsf{eval}(\bar{t}) \wedge \forall \bar{y}.\ \mathsf{eval}(\bar{y}) \rightarrow (F(\mathsf{eval}))(\bar{y})$.

Because νF is the *greatest* pre–fixed–point, for any pre–fixed–point S of F, $G' \in |S(\bar{t})|$ implies $G' \in |(\nu F)(\bar{t})|$. This means that $G' \models (\nu F)(\bar{t})$.

This proof relies on two lemmas, which we will describe in turn. The first proves that propositions of a certain form are pre–fixed–points of F, and the second establishes that, given specifications in the grammar of Eqn. 2, we can compute by exhaustive forward reasoning databases that meet the specification.

Lemma 2. *If F is a substitution function $\lambda P.\lambda \bar{x}.P_f(\bar{x})$ and the predicate* eval *does not appear in P', then $S = \lambda \bar{x}.\ \mathsf{eval}(\bar{x}) \wedge \forall \bar{y}.\ \mathsf{eval}(\bar{y}) \rightarrow (F(\mathsf{eval}))(\bar{y})$ is a pre–fixed–point of F.*

Proof. Given an arbitrary $\bar{t}$ and a database G such that $G \models S(\bar{t})$, we must show $G \models (FS)(\bar{t})$. By definition, $G \models \mathsf{eval}(\bar{t})$ and $G \models \forall \bar{y}.\ \mathsf{eval}(\bar{y}) \rightarrow (F(\mathsf{eval}))(\bar{y})$, and therefore $G \models (F(\mathsf{eval}))(\bar{t})$. We then proceed, as in Lemma 1, by induction on the structure of P_f to prove that $(F(\mathsf{eval}))(\bar{t}) \Rightarrow (FS)(\bar{t})$. The only interesting case is when we reach the substitution and have $G \models \mathsf{eval}(\bar{s})$ and must prove $G \models S(\bar{s})$. But this only requires that we prove $G \models \mathsf{eval}(\bar{s})$ and $G \models \forall \bar{y}.\ \mathsf{eval}(\bar{y}) \rightarrow (F(\mathsf{eval}))(\bar{y})$, which we have already proved.

Lemma 3. *If the proposition P is run as an a forward reasoning logic program starting with the initial database G, and the result is a saturated database G', then $G' \models P$.*

Proof. By contradiction and by induction on the structure of P. If it is not the case that $G' \models P$, then G' is not in fact a saturated database.

3 Specifying 0CFA

Now we can give a coinductive specification of a control flow analysis in the style of Nielson et al. and describe the forward–chaining logic program that

implements the specification. We do not have space to relate a coinductive specification to the 0CFA analysis derived from an operational semantics in [4], but this example illustrates the issues. We are interested in control flow analysis for a language of labeled terms from the lambda calculus. Variables are represented as constants (as opposed to a higher–order abstract syntax representation).

$$e ::= t^l \qquad\qquad t ::= \mathsf{app}(e_1, e_2) \mid \lambda(x, e) \mid x \tag{6}$$

Straightforwardly transcribing 0CFA as it appears in [2] or [6] is not particularly illustrative:

$$\begin{aligned}
\nu\lambda P.\ \lambda e.\ \ & \forall x.\forall l.\ e \doteq x^l \to \mathsf{bind}(x, v) \to \mathsf{return}(l, v) \\
& \wedge \forall x.\forall e.\forall l.\ e \doteq \lambda(x, e_0)^l \to \mathsf{return}(l, \lambda(x, e_0)^l) \\
& \wedge \forall t_1.\forall l_1.\forall t_2.\forall l_2.\forall\ e \doteq \mathsf{app}(t_1^{l_1}, t_2^{l_2})^l \to \\
& \qquad P(t_1^{l_1}) \wedge P(t_2^{l_2}) \\
& \qquad \wedge\ \forall x.\forall t.\forall l_0.\ \mathsf{return}(l_1, \lambda(x, t^{l_0})) \to \\
& \qquad\quad (\forall v_2.\ \mathsf{return}(l_2, v_2) \to \mathsf{bind}(x, v_2)) \\
& \qquad\quad \wedge\ P(t^{l_0}) \wedge (\forall v_0.\ \mathsf{return}(l_0, v) \to \mathsf{return}(l, v)
\end{aligned} \tag{7}$$

However, the corresponding logic program, after being transformed to remove equalities, factored by introducing the new predicate comp, and flattened into the Horn fragment, begins to resemble the derived 0CFA in [4].

$$\begin{aligned}
&\mathsf{eval}(x^l) \to \mathsf{bind}(x, v) \to \mathsf{return}(l, v) \\
&\mathsf{eval}(\lambda(x, e)^l) \to \mathsf{return}(l, \lambda(x, e)) \\
&\mathsf{eval}(\mathsf{app}(t_1^{l_1}, t_2^{l_2})^l) \to (\mathsf{eval}(t_1^{l_1}) \wedge \mathsf{eval}(t_2^{l_2}) \wedge \mathsf{comp}(l, l_1, l_2)) \\
&\mathsf{comp}(l, l_1, l_2) \to \mathsf{return}(l_1, \lambda(x, t^{l_0})) \to \mathsf{return}(l_2, v_2) \to \mathsf{bind}(x, v_2) \\
&\mathsf{comp}(l, l_1, l_2) \to \mathsf{return}(l_1, \lambda(x, t^{l_0})) \to \mathsf{eval}(t^{l_0}) \\
&\mathsf{comp}(l, l_1, l_2) \to \mathsf{return}(l_1, \lambda(x, t^{l_0})) \to \mathsf{return}(l_0, v) \to \mathsf{return}(l, v)
\end{aligned}$$

The differences are partly a manner of program transformation, but they are also partly a result of the fact that the specification of an acceptable 0CFA analysis in Nielson et al. is slightly less precise than the derived one by Simmons and Pfenning.

References

1. D. A. McAllester. On the complexity analysis of static analyses. *J. ACM*, 49(4):512–537, 2002.
2. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.
3. F. Nielson, H. Seidl, and H. R. Nielson. A Succinct Solver for ALFP. *Nord. J. Comput.*, 9(4):335–372, 2002.
4. R. J. Simmons and F. Pfenning. Linear logical approximations. In G. Puebla and G. Vidal, editors, *PEPM*, pages 9–20. ACM, 2009.
5. D. Van Horn and H. G. Mairson. Relating complexity and precision in control flow analysis. In *ICFP '07*, pages 85–96, New York, NY, USA, 2007. ACM.

6. D. Van Horn and H. G. Mairson. Deciding kCFA is complete for EXPTIME. In *ICFP '08*, pages 275–282, New York, NY, USA, 2008. ACM.
7. D. Van Horn and H. G. Mairson. Flow analysis, linearity, and PTIME. In *SAS*, volume 5079 of *LNCS*, pages 255–269. Springer, 2008.