Mechanically Checking Invariants of Linear Logic Programs

Chris Martens

July 28, 2014

Abstract

This note describes progress toward an algorithm for checking properties of linear logic programs described with *generative signatures*, which themselves are linear logic programs.

1 Introduction

Linear logic has been used fruitfully as a specification language for describing stateful, concurrent, distributed, and interactive computation. Within a logic programming framework, we can execute these specifications and reason about their operational semantics.

Our broader goal is to be able to reason about these operational semantics as fully as we can reason about other stateful programming approaches. The specific technique we investigate here prescribes a workflow wherein the programmer describes a desired property of their program, and the property may be checked as an invariant or postcondition. The language for describing properties, i.e. the language of *generative signatures*, is a proper subset of the language in which programs are themsevles written.

2 Generative Signatures

Generative signatures are like grammars for linear contexts. For example, the signature

```
gen -o {a * gen}.
gen -o {1}.
```

equipped with the *seed context* {`gen`}, describes contexts containing zero or more instances of `a`. Formally, the set of context a signature Σ and a seed Δ_0 describes is the set of reachable contexts from Δ_0 following rules in Σ.

Generative signatures were first discussed in Rob Simmons's thesis (XXX cite). There they were specifically being used as invariants, so he referred to them as generative *invariants*.

3 Notation

In `verbatim` text we'll use `-o` for the lolli symbol, which we write as $\multimap$ in math text. Similarly, verbatim `*` is tensor, or $\otimes$.

We're sticking to a multiplicative, atomic-propositional fragment of LL for now. We are also only considering rules of the form $A \multimap B$, where neither A nor B contain $\multimap$ (the "first order" fragment, in the logic programming sense of the word).

We'll write p to stand for a generic propositional atom.

4 Problem Setup

Here's a linear logic program.

```
a * a -o {a}.
a * b -o {a * b * b}.
```

With a program this small, we can reason informally about some of its properties. For instance, if there are no `a`s in the context, the program cannot evolve the context further. We can also reason that if there is an `a` and a `b`, the program will never terminate. This deduction uses two properties of the program: first, that the second rule will continue to produce `b`s as long as there is an `a`, and second, that the existence of at least one `a` in the context is an invariant of the program (as long as we start out with one).

We can characterize these kinds of properties with generative signatures. For instance, if we wanted to describe the "at least one `a`" property as a generative signature, we might try this slight modification of the generative signature from the previous section:

```
gen -o {a * gen}.
gen -o {a}.
```

To check this signature (let's call it Σ_{gen}) as an invariant of the original program (let's call it Σ), we need to check that (a) it holds of whatever initial context we specify along with Σ and (b) that each rule in Σ *preserves* the invariant. The former problem is a subset of the latter, so we will focus on the latter: invariant preservation on a per-program-rule basis.

4.1 Synthetic Transitions

There's just one more piece we need before we can define rule preservation precisely. That is the relationship between rules $A \multimap B$ and the operational semantics of the program, which can be understood in terms of *transitions* between contexts $\Delta \leadsto \Delta'$. (To be precise, we often write $\Delta \leadsto_\Sigma \Delta'$ to indicate which program Σ we are stepping along.)

The common case of how rules are related to operational semantics is that a rule $a_1 \otimes \ldots \otimes a_n \multimap b_1 \otimes \ldots \otimes b_m$ induces a *synthetic transition*

$$\Delta, a_1, \ldots, a_n \leadsto \Delta, b_1, \ldots, b_m$$

which is parametric over Δ.

In fact, *every* transition the program makes per its original definition as proof search in a focused logic can be superceded by these synthetic transitions. A full account of this correspondence can be found in (XXX cite DeYoung).

We will write A^* to mean

$$\begin{array}{rcl} p^* & = & p \\ (A \otimes B)^* & = & A^*, B^* \end{array}$$

to transform tensors of atoms into contexts.

4.2 Generative Property Preservation

Now we can define what it means for a rule to preserve a generative property.

Definition 1: a rule $r : A \multimap B$ preserves a generative property $\langle \Sigma_{gen}, \Delta_0 \rangle$ iff for all Δ, whenever $\Delta_0 \leadsto_{\Sigma_{gen}} \Delta, A^*$, it is also the case that $\Delta_0 \leadsto_{\Sigma_{gen}} \Delta, B^*$.

How the first rule passes: Upon inspection of the first rule in Σ, $A = a \otimes a$ and $B = a$. This means we have by assumption that for all Δ, gen $\leadsto_{\Sigma_{gen}} \Delta, a, a$ and we need to show gen $\leadsto_{\Sigma_{gen}} \Delta, a$. Intuitively, this means reasoning about how the two as were generated in the first production and recognizing that the second generative signature rule must have been applied last (assuming no instances of gen remain in Δ, which we should explicitly state as an assumption). Since applying that rule ends the generative computation, it can only have been applied once, meaning the other a must have come from the first generative rule, preserving gen. And the application of that rule could instead be replaced by the second rule (which fires on the same premises), decreasing the number of as produced by one.

This handwavy argument can be made more formal by induction over the length of the trace and reasoning by permutative equality; see (XXX cite Rob's thesis) for details.

How the second rule passes – but not quite how we wanted!: Note that if we try to reason about how our generative signature creates a context Δ, a, b, we fail – because we haven't given any rules that can produce b. Thus the *implication* succeeds, but only vacuously. If we seeded our program with any bs, the base case for the proof would fail, and for any other seed context, the program rule would never fire.

Let's modify the generative signature to *fully* characterize the set of contexts that we expect.[1]

```
gen -o {a * gen}
gen -o {a * bs}.
bs -o {1}.
bs -o {b * bs}.
```

Note: we might expect the algorithm to flag these "cannot fire" rules even if they don't violate the invariant.

In this extended context, we can reason about how Δ, a, b was produced with the same technique of induction over the production trace, then inject another instance of the fourth generative rule to arrive at the well-formation of Δ, a, b, b.

[1]Future work includes consideration for *modular* specifications, i.e. properties that can be specified as only partial representations of the context.

5 Analogy with Grammars

In some sense, these generative signatures can be read as production rules for *grammars* of multisets. That is, we can think of a linear context Δ as a multiset of atoms and ask whether it can "parse" as a particular seed atom, such as gen, much like we can ask whether a given string or sequence of part-of-speech tokens parses as a sentence. For this reason, we will sometimes write $\Delta \in \mathsf{L}(\Sigma_{gen}, \mathsf{gen})$ for $\mathsf{gen} \rightsquigarrow_{\Sigma_{gen}} \Delta$.

We also borrow the words "terminal" and "nonterminal" from grammars, i.e. we will call an atom that only appears on the right hand side of rules *terminal* and an atom that may appear to the left as *nonterminal*.

In order for the context language to be "regular," i.e., writeable as a finite state machine, states must be mapped to nonterminals, and we must always be in just one state. This means rules have the form

$$n \multimap \{t_1 \otimes \ldots \otimes t_n(\otimes n')\}$$

where n is a nonterminal, t_i are terminals, and $(\otimes n')$ is an optional nonterminal.

5.1 Epsilon Transitions

Some algorithms over grammars, e.g. for parsing, are easier to implement if they don't have to deal with rules corresponding to unlabeled state machine transitions, i.e. that don't emit/consume any terminal atom. Grammars with these "epsilon transitions" can (usually?) be normalized into ones that don't have them (via what algorithm? cite something). For example, the grammar described by the signature

```
gen -o {a * cs}
cs -o {c * cs}
c -o {1}
```

where the last rule is an epsilon transition, can be transformed into

```
gen -o {a}
gen -o {a * cs}
cs -o {c * cs}
cs -o {c}
```

where the choice to emit 0 cs is "pushed back" further toward the beginning of the trace.

It is currently unknown whether we *need* to normalize signatures this way, but it seems to make things easier in a small number of case we've considered, so for the remainder of the document we'll assume epsilon-transition-free generative signatures.

6 The checking algorithm, first take

At a high-level approximation, we determine whether a generative signature Σ_{gen} is preserved by a rule $r : A \multimap \{B\}$ by first performing a process of *inversion* on the supposition that $gen \rightsquigarrow \Delta, A^*$, then using the facts learned by way of inversion to demonstrate $gen \rightsquigarrow \Delta, B^*$.

The most complex core of this algorithm is inversion.

6.1 Inversion

Inversion is generalized to take a pair of contexts Θ_1 and Θ_2 (instantiated at gen and Δ, A^* at the top level), which then *splits* the supposed trace between those contexts along a rule that could produce a known atom in A^*.

By convention we write Θ for contexts that may contain nonterminals.

Coframe property: If t is terminal and we have $\epsilon : \Theta, t \leadsto_{\Sigma_{gen}} \Theta', t$, then $\epsilon : \Theta \leadsto_{\Sigma_{gen}} \Theta'$. **Proof** by inspection of the definition of "terminal" and possible formations for ϵ: no rules in Σ_{gen} can consume t, so no steps of ϵ will depend on it.

This observation allows us to recursively apply inversion on the right-hand split of the trace and ensure termination (we always subtract an atom from the right-hand side and work by inversion on a new one).

This property also allows us to invert on the atoms in A^* in arbitrary order.

Inversion trees and paths: Because multiple rules may apply that give rise to the atom along which we perform inversion, we maintain a tree structure to hold all possible inversion paths. The following recurisve datatype is used to represent the output of inversion:

```
datatype invTree = Leaf | Node of (trace * trace * invTree) list
```

Nodes contain a list, one for each rule that applies, of splittings—the left and right half of the split (each pairs of contexts between which there exists a trace), and the child tree that represents the remainder of the computation.

All inversion paths can be enumerated by enumerating paths through the tree. For each path, we'll need to try adding the inversions to the generative signature to prove the program rule's consequence from gen using the additional information learned through inversion. Every such test will need to succeed for the rule to be sound.

The ML code implementing inversion can be found in Figure 1.

Incompleteness This algorithm only recursively splits the right half of the trace. This is because we don't yet know how to make it decideable to split the left. It succeeds anyway on a fair number of cases, but there are some pathological ones (which we describe later) where it fails.

7 Correctness Criteria

Correctness of the algorithm means that iff it says yes, then every rule in the program preserves the generative property. By extension, iff it says yes to a given rule $r : A \multimap \{B\}$, then $gen \leadsto_{\Sigma_{gen}} A$ implies $gen \leadsto_{\Sigma_{gen}} B$.

Correctness might fail, in terms of the inversion algorithm, for several reasons. Because every inversion path must succeed, it may be the case that we generate too many or too few paths. Alternatively, we might split paths in invalid ways, such that the extended signature checks the RHS's membership under invalid assumptions.

We also need to make sure that the extended generative signatures stay within a decidable fragment. They will no longer be in the state machine fragment, but it seems like it should still be at worst a "petri net"- like fragment, in which case the decideability of reachability for petri nets bodes well.

8 A pathological case

Consider the generative signature $\Sigma_{gen} =$

```
g1 : gen -o {a * cs}.
g2 : gen -o {b * c}.
g3 : cs -o {c * cs}.
g4 : cs -o {1}.
```

Σ_{gen} has an "epsilon transition," that is, a rule which consumes a nonterminal without producing a terminal (g4). This causes some issues in the inversion algorithm, so we normalize the signature as follows. $\Sigma'_{gen} =$

```
g1 : gen -o {a}.
g2 : gen -o {a * cs}.
g2 : gen -o {b * c}.
g3 : cs -o {c * cs}.
g4 : cs -o {c}.
```

Note that the class of contexts (Σ'_{gen}, gen) describes is one where an a can appear alongside arbitrarily many cs, but a b requires *exactly one* c alongside it.

Consider checking the validity of these two rules:

```
(1) c * c -o {1}
(2) c -o {1}
```

The first rule should succeed – if there are two cs in the context, then we must have generated them through a cs, which came with an a – which we can preserve by emitting no cs at all. But the second rule should fail, because if it was emitted alongside a b, then we will not be able to recover the b from an equivalent context generated with no cs.

RHS-inversion only fails to capture the constraints on the context necessary to perform this reasoning.

In the case of rule (1), it seems that we need to do some analysis of the LHS of the trace to *rule out* the use of g2 for generating Δ, c, c. This reasoning depends on the fact that gen cannot spontaneously regenerate, nor can there be multiple copies of it at any point in the generative trace. (Interestingly, this property could *itself* be described with a generative signature, though I doubt that kind of metacircularity will prove helpful.)

9 Desired Eventual Scope

Eventually, we'd like to be able to give generative invariants that

- include properties of first-order predicates over finite datatypes (e.g. on(X,Y) from the Blocks World domain)
- can describe not just *invariants* but also *quiescence* properties of contexts that are the result of a program's execution when no more rules can fire, and conversely, *activity* properties (the complement of quiescence)
- less certainly, we might wish to describe properties of programs outside the "horn fragment," i.e. programs with implicational subgoals ($(A \multimap B) \multimap C$).

```
(* invs : gensig -> trace -> invTree option *)
  fun invs S (lhs, nil) = SOME Leaf
    | invs S (lhs, a::D) =
        let
          val rules = List.filter (fn (bs, ays) => member a ays) S
        in
          case rules of
               nil => NONE
             | _ =>
               let
                 val theta = gensym()
                 fun split (bs, ays) =
                   ((lhs, theta::bs),
                    (theta::(deleteFirst a ays), D))
                 fun child r =
                    let
                      val (ltrace, rtrace) = split r
                      val result = invs S rtrace
                    in
                      case result of
                          NONE => NONE
                        | SOME tree => SOME (ltrace, rtrace, tree)
                    end
                 fun filterNones (NONE::l) = filterNones l
                   | filterNones ((SOME x)::l) = x::(filterNones l)
                   | filterNones [] = []
                 val children = filterNones (map child rules)
               in
                 case children of
                      [] => NONE
                    | _ => SOME (Node children)
               end
        end
```

Figure 1: ML code implementing (right-only) inversion