Mechanically Checking Invariants of Propositional Linear Logic Programs

Chris Martens

September 30, 2014

Abstract

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. However, very little is known about *mechanically reasoning* about them.

This note describes a decidability proof and algorithm for checking properties of the *propositional Horn fragment* of forward-chaining linear logic programs.

The properties are described with *generative signatures*, which themselves are linear logic programs of an even more limited fragment—a class whose reachability problem is known to be expressible in Presburger arithmetic. Using that expressibility result, we reformulate the invariant preservation problem as a Presburger arithmetic proposition.

Finally, we show how the intermediate representation of a generative signature giving rise to the Presburger formula makes it more amenable to a more direct approach in terms of linear logic proof terms.

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; however, little is known about how to reason about their operational semantics.

The technique we investigate for program reasoning prescribes a workflow wherein the programmer describes a desired property of their program, and the property may be checked as an invariant. The language for describing properties, i.e. the language of *generative signatures*, is a proper subset of the language in which programs are themselves written.

We describe an algorithm for checking the *propositional Horn fragment* of linear logic programs under a class of generative signatures corresponding with *flatable languages*, which have been previously shown expressible in Presburger Arithmetic, a decidable theory.

1.1 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 as 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 bs 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.

1.2 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.

1.3 Paper Organization

The remainder of the paper is organized as follows:

2 Background

2.1 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* $\{\texttt{gen}\}$, describes contexts containing zero or more instances of $\texttt{a}$. 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 [**?**]. There they were specifically being used as invariants, so he referred to them as generative *invariants*.

2.2 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 [**?**].

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.

2.3 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, $\mathsf{gen} \leadsto_{\Sigma_{gen}} \Delta, a, a$ and we need to show $\mathsf{gen} \leadsto_{\Sigma_{gen}} \Delta, a$. 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 $\texttt{gen}$ 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 $\texttt{gen}$. 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 [**?**] 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.

2.4 A Pathological Example

Consider the following generative signature.

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

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.

Rules that should pass (preserve the invariant):

1. `a -o {a * c}.`
2. `a * c -o {a}.`

That is, if there is an a in the context, we should be able to add or subtract arbitrarily many cs.

Rules that should fail (do not preserve the invariant):

1. `1 -o {c}.`
2. `c -o {1}.`

That is, we may not arbitrarily remove or generate a c. The second generative rule g2 permits a context with *exactly* one b and two cs.

Thus, our algorithm needs to be able to account for preservation not on a purely local, or context-free, basis, but rather in a sense that takes into account *all the possible ways* an atom on the left-hand side of the rule could be generated—which may impose some constraints on the context—and conclude that, in all of those scenarios, including their constraints, the right-hand side of the rule could have been generated in its place.

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

3 Methodology

3.1 False Start: Trace Splitting

The first thing we tried was to 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*.

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 $\epsilon : \Theta, t \rightsquigarrow_{\Sigma_{gen}} \Theta', t$, then $\epsilon : \Theta \rightsquigarrow_{\Sigma_{gen}} \Theta'$. **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 **??**.

IncompletenessThis 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.

3.2 Restating the problem in terms of Vector Addition Systems

In general, we cannot program a computer to reason inductively about arbitrary program traces such that it always terminates. In this section we describe how to model generative signatures and programs as *vector addition systems* in a way that they can be treated using known techniques. Specifically, we show how invariant preservation can be modeled in Presburger Arithmetic, the first-order theory of natural numbers with inequality and addition, which happens to be decidable.

A *vector addition system* (VAS) V is an initial configuration $\vec{x_0} \in \mathbb{N}^n$ for a fixed dimension n, along with a set T of *transitions* describing how a configuration may evolve. Transitions can be described as vectors $\vec{t} \in \mathbb{Z}^n$ along with side conditions of the form $\vec{k} \in \mathbb{N}^n$ imposing the inequality $\vec{v} \geq \vec{k}$.

If the conditions hold of a configuration $\vec{v}$ (i.e. $\vec{v} \geq \vec{k}$) *and* $\vec{v} + \vec{t} \geq \vec{0}$, the transition $\vec{t}$ is said to be *applicable* or *fireable*, and the transition relation $\rightsquigarrow$ holds of $\vec{v} \rightsquigarrow \vec{v} + \vec{t}$.

Typically, for these systems, which are equivalent to Petri nets (XXX cite), the question is whether the *reachability* relation, or transitive transition relation, holds between two configurations. More generally, one might want to compute the *set* of reachable configurations from a given set of configurations S (along a set of transitions T). Such a set is called the *reachability set* of S (along T).

A correspondence is known (XXX cite kanovich?) between reachability for vector addition systems and provability of a sequent in the propositional Horn (or Petri net) fragment of linear logic. Simply put, a VAS transition rule of the form $(\vec{t}, \vec{k})$ can be interpreted as a linear logic rule

$$a_1^{c_1} \otimes \ldots \otimes a_n^{c_n} \multimap \{a_1^{c_1+t_1} \otimes \ldots \otimes a_n^{c_n+t_n}\}$$

Where $c_i = \mathsf{max}(k_i, -t_i)$ and a^c means $a \otimes \ldots \otimes a$ with c repetitions.

Computing reachability sets is known to be decidable. (XXX cite something.) However, the decidability of this problem does not help us directly with deciding if a given program rule preserves a generative invariant. Cast as a VAS problem, the question is whether

For all configurations c_A such that $r : A \multimap \{B\}$ is fireable, if $\vec{c_0} \rightsquigarrow^* \vec{c_A}$ then $\vec{c_0} \rightsquigarrow^* \vec{c_A} + tk_r$ (where $tk_r = (t_r, k_r)$ is the transition and condition vector corresponding with the rule r).

The *postset* of a given vector set S along a rule t, denoted $\mathsf{post}_{(t,k)}(S)$, is the set of all vectors v' such that there exists a $v \in S$ where (t, k) is fireable on v and $v' = v + t$.

So we can reword the above criterion for preservation as simply

$$\mathsf{post}_{(tk_r)}(S) \subseteq S$$

where S is the reachability set of c_0 (the initial configuration of the generative invariant) along the rules of the generative invariant.

In general, computing the reachability set does not help us answer this question. But if the reachability set corresponds to a Presburger formula, then the entire thing can be expressed as a Presburger formula, and Presburger arithmetic is decidable. Thus, what remains is to show that the particular subset of VASs that correspond with generative invariants are Presburger.

3.3 Presburger Vector Addition Systems

In general, VASs are not expressible as Presburger Arithmetic formulae. (XXX cite something) Briefly speaking, one can encode exponentiation as a VAS, which is not Presburger. A full example (from XXX cite) is (XXX give example).

In order to show that generative invariant preservation is decidable, we need to isolate a fragment of the VAS language that is both suitable to use for our generative invariants of interest *and* expressible in Presburger Arithmetic.

We accomplish this task by using prior work (XXX cite) that demonstrates that Presburger-expressible VASs are exactly those which have an equivalent "flat" representation. A *flat* VAS is one in which every run (sequence of applicable rules) is in the grammar

$$(w_1)^* \ldots (w_n)^*$$

for some finite set of *words* (finite transition sequences) $w_1 \ldots w_n$.

A *flatable* VAS is one with the same reachability set as a flat VAS. One known class of flatable VASs are so-called Basic Parallel Processes, or BPPS (XXX cite), which are those corresponding to sets of Horn linear logic rules with a single premise (or Petri nets where every transition has a single input). Although this class may sound limiting, every generative invariant we have studied so far meets this criterion. [2]

There exist published, proven terminating, and implemented methods to decompose VASs such as BPPs into flat languages. (XXX cite olsen, fribourg) These techniques involve iteratively disentangling cyclic rule dependencies. As an alternative to automatically finding an equivalent flat language, we can simply stipulate that a given generative invariant must be flat.

3.4 Flat(able) Generative Invariants

Here is a generative invariant corresponding loosely to a propositional erasure of a well-formedness specification for Blocks World. (XXX mention blocks world earlier?)

```
g1 : gen -o {t * gen'}.  %% t ~= "on table"; construct a new stack
g2 : gen' -o {b * gen'}. %% b ~= "on"; add a block to a stack
g3 : gen' -o {c * gen}.  %% c ~= "clear"; finish a stack and return
g4 : gen -o {f}.         %% f ~= "arm free"
g5 : gen -o {h}.         %% h ~= "arm holding a block"
```

This specification is not flat: there is "nested" looping of the rule g2 within the wider loop formed by g1 and g3. This introduces the constraint that a t and c both *must* be generated in order for there to be more than zero bs. It also means that, if there are no nonterminals gen and gen', the number of ts and cs must be the same.

As a VAS, these rules correspond to transitions (XXX)

The output of the automatic decomposition technique on this VAS is

Alternatively, a hand-crafted flat version of this specification is as follows.

```
g1 : gen -o {t * gen * gen'}  %% begin a new stack
g2 : gen' -o {b * gen'}.      %% add a block to a stack
g3 : gen' -o {c}.             %% finish a block stack
g4 : gen -o {f}.
g4 : gen -o {h}.
```

This signature allows multiple block stacks to be created concurrently. It cannot be interpreted as a state machine; the rule g1 introduces the possibility of being in multiple

[2]Simmons' examples for the operational semantics of programming languages (XXX cite rob thesis) occasionally include a second premise, but always a persistent one (premise of the form $!A$). This kind of rule may still be expressible as a BPP so long as the persistent premise can be represented as a side condition, but we currently exclude these examples from our proofs.

"states" simultaneously. Nonetheless, it is still a BPP (one premise per rule), and it is already flat. The flat language representing the reachability set is

$$(g_1)^*(g_2)^*(g_3)^*(g_4)^*(g_5)^*$$

(XXX note interesting things about the relationship between these signatures, parallelism, cps, ?)

4 Results

4.1 Computing Presburger reachability sets

(XXX cite leroux?)

(XXX define the "weakest precondition" config of a word, c_w)

In general, the postset of a vector set under an iterated word w^* can be given by binding an existential variable for the number of iterations, i.e.

$$\mathsf{post}(S, w^*) = \{x \mid \exists v \in S.\exists n \in \mathbb{N}.(XXX) x = v + na \wedge v \geq c_w\}$$

For a VAS with starting configuration c_0, a sequence of Presburger sets C_i is computed inductively:

$$\begin{aligned} C_0 &= \{c_0\} \\ C_i &= \mathsf{post}(C_{i-1}, (w_i)^*) \end{aligned}$$

4.2 End-to-End Example

Let's revisit the pathological case from (XXX reference section).

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

As a vector addition system over `<gen, cs, a, b, c>`:

```
g1 : <-1,  1, 1, 0, 0>   ; <1, 0, 0, 0, 0>
g2 : <-1,  0, 0, 1, 2>   ; <1, 0, 0, 0, 0>
g3 : < 0,  0, 0, 0, 1>   ; <0, 1, 0, 0, 0>
g4 : < 0, -1, 0, 0, 0>   ; <0, 1, 0, 0, 0>
```

As a flat language:

```
g2*g1*g3*g4*
```

As a more precise flat language:

```
(g2 | (g1(g3*)g4))
```

As a Presburger formula specification:

$$C = C_{1A} \cup C_{3B}$$

where

$$\begin{aligned} C_0 &= \{\texttt{<1, 0, 0, 0, 0>}\} \\ C_{1A} &= \mathsf{post}(C_0, \texttt{g2}) \\ C_{1B} &= \mathsf{post}(C_0, \texttt{g1}) \\ C_{2B} &= \mathsf{post}(C_{1B}, \texttt{g3*}) \\ C_{3B} &= \mathsf{post}(C_{2B}, \texttt{g4}) \end{aligned}$$

The only piece of this formula requiring a quantifier is the repeated g3*. The reachability set involving g3* can be described as:

$$\begin{aligned} C_{2B} &= \mathsf{post}(C_{1B}, \texttt{g3*}) \\ &= \{x' \mid \exists n.\texttt{<0,1,1,0,0>} + n\texttt{<0,0,0,0,1>} = x'\} \\ &= \{x' \mid \exists n.x' = \texttt{<0,0,1,0,n>}\} \end{aligned}$$

Thus the final solution for the original GI as a Presburger formula is:

$$C = \{\texttt{<0,0,0,1,2>}\} \cup \{\vec{x} \mid \exists n.\vec{x} = \texttt{<0,0,1,0,n>}\}$$

Checking Program Rules

Now we can work out the decision procedure for the example on a couple of different program rules. We revisit two examples from section (XXX ref pathological example section), one that preserves the invariant and one that does not:

1. `a -o {a * c}.` (should pass)
2. `1 -o {c}.` (should fail)

4.3 Adequacy and Correctness

Theorem statement (adequacy): modeling generative invariants as vector addition systems is sound and complete with respect to derivability and reachability. That is, formally, if $\mid \Sigma_{gen}, \Delta_0 \models \langle V, c_0 \rangle$ then $\Delta_0 \rightsquigarrow^*_{\Sigma_{gen}} \theta$, if and only if $\mid \theta \models x$ and $c_0 \to^*_V x$.

Theorem statement (soundness): if the Presburger formula representing a rule $r : A \multimap \{B\}$ preserving a generative invariant $\langle \Sigma, \Delta_0 \rangle$ has a proof, then

$$\Delta_0 \rightsquigarrow^*_\Sigma A$$

if and only if

$$\Delta_0 \rightsquigarrow^*_\Sigma B$$

Theorem statement (completeness): converse of soundness.

5 Revisiting Trace Splitting

A generative trace

$$\epsilon : \mathsf{gen} \rightsquigarrow^* \delta, t_1, \ldots, t_n$$

within the "pathological case" generative signature must inhabit the flat language that characterizes it:

```
(g2 | (g1(g3*)g4))
```

This means that when we invert on this trace to split it into smaller pieces, we have more information than previously.

6 Conclusion

```
(* 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