Mechanically Checking Invariants of Propositional Linear Logic Programs

Chris Martens

October 3, 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 on paper, much less algorithmically.

We present a technique for reasoning automatically about linear logic programs. This technique enables the programmer to describe a desired invariant, which can then be checked automatically. Instead of developing a new meta-language over linear logic programs, we propose the use of *generative signatures* as an interface to writing environments, where generative signatures are simply a restricted subset of Horn linear logic programs.

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 Example

The overarching goal of this project is to be able to specify and automatically check properties of programs that manipulate resources. As a simple example, consider a program to simulate coin exchange: a dime can be traded for two nickels, a quarter can be traded for two dimes and a nickel. This specification can be given in linear logic as two rules:

$$\begin{array}{lcl} r1 & : & d \multimap n \otimes n \\ r2 & : & q \multimap d \otimes d \otimes n \end{array}$$

In concrete notation borrowed from Celf (XXX), we write:

```
r1 : d -o {n * n}.
r2 : q -o {d * d * n}.
```

1.2 Linear Logic Programming

celf, notation, etc (XXX)

1.3 Paper Organization

The remainder of the paper is organized as follows: Section 2 provides the formal definitions for the fragment of linear logic from which we consider programs, generative signatures, and preservation of such signatures by a program. Section 3 describes our approach, including the "false start" of iteratively splitting program traces, then moves on to desribe an encoding of the problem via Vector Addition Systems and Presburger Arithmetic. In Section 4 we give the decidability proof and algorithm via compiling the question of invariant preservation into a Presburger formula. We then walk through an example and prove correctness of the algorithm. Finally, in Section 5, we discuss ongoing work on extending the procedure to handle the first-order case. We revisit the "false start" idea of trace-splitting in light of the Presburger formula extraction approach and suggest its use for handling this case.

2 Background

2.1 The Propositional Horn Fragment of Linear Logic

2.2 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 [7]. There they were specifically being used as invariants, so he referred to them as generative *invariants*.

2.3 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 $\Delta \rightsquigarrow \Delta'$. (To be precise, we often write $\Delta \rightsquigarrow_\Sigma \Delta'$ 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*

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

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 [1].

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.

2.4 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 $\Delta_0 \rightsquigarrow_{\Sigma_{gen}} \Delta, A^*$, it is also the case that $\Delta_0 \rightsquigarrow_{\Sigma_{gen}} \Delta, B^*$.

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, $\mathsf{gen} \rightsquigarrow_{\Sigma_{gen}} \Delta, a, a$ and we need to show $\mathsf{gen} \rightsquigarrow_{\Sigma_{gen}} \Delta, a$. 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 [7] 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 `b`s, 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]

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

```
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.

2.5 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 `c`s, 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 `c`s.

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 `c`s.

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.

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 $gen \leadsto \Delta, A^*$, then using the facts learned by way of inversion to demonstrate $gen \leadsto \Delta, B^*$.

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 $\epsilon : \Theta, t \rightsquigarrow_{\Sigma_{gen}} \Theta', t$, 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 3.1.

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.

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* (first introduced by Karp and Miller [3]) 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 (see e.g. [4]), the question is whether the *reachability* relation, or transitive transition relation, holds between two

```
(* 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

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 [2] 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 for vector addition systems is known to be decidable. [6] 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 $\vec{c_0} \leadsto^* \vec{c_A}$ then $\vec{c_0} \leadsto^* \vec{c_A} + tk_r$ (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. An explanation of this fact and a characterization of the Presburger fragment is given in [5]. Briefly speaking, one can encode exponentiation as a VAS, which is not Presburger. In this section, we recapitulate Leroux's characterization of Presburger Vector Addition Systems in terms of how we use them for generative invariants.

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.

Leroux shows that the 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

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

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

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.

[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.

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 `b`s. It also means that, if there are no nonterminals `gen` and `gen'`, the number of `t`s and `c`s 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 "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

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.

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

where c_w is XXX define

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 The Algorithm

$$|(\Sigma, \Delta_0)| \quad = \quad ... \tag{1}$$

Now that we have a Presburger characterization of the reachability sets of generative invariants, we can describe what it means for a rule $r : A \multimap \{B\}$ to preserve an invariant (Σ, Δ_0), as a Presburger formula itself:

Let $\mathsf{post}^*_{|\Sigma|}(\Delta_0) = S$. The rule r preserves Σ iff

$$\mathsf{post}_{|r|}(S) \subseteq S$$

4.3 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)

Consider the postset of C along each rule: for rule 1, the only part affected is the second disjunct. $C' = \mathsf{post}_1(C) =$

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

But this last unioned set is a subset of $\{\vec{x} \mid \exists n.\vec{x} = \texttt{<0,0,1,0,n>}\}$ (which can be determined in Presburger Arithmetic by modeling subsethood as implication).

For rule 2, $C'' = \mathsf{post}_2(C) =$

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

This set is not a subset of the original, so it fails, as expected.

4.4 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 $\mid \Sigma_{gen}, \Delta_0 \mid= \langle V, c_0 \rangle$ then $\Delta_0 \rightsquigarrow^*_{\Sigma_{gen}} \theta$, if and only if $\mid \theta \mid= x$ and $c_0 \rightarrow^*_V x$.

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 Discussion

5.1 Toward the First-Order Case

5.2 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.

5.3 Additional Future Work

XXX meld programs; voting protocol
quiescence states; initial states; liveness conditions
integration with phases

5.4 Conclusion

We have given a decidability proof and decision procedure for determining whether a Horn propositional linear logic program preserves an invariant given by a generative signature. We have shown that this problem can be encoded in Presburger arithmetic via flattening generative invariants. Flattening is always a terminating process for the generative invariants we have considered so far.

This work represents a very preliminary investigation into the automatic checking of properties described by generative invariants. While it demonstrates that the idea passes the initial litmus test of decidability, compiling to, and checking, a Presburger formula would likely not be computationally efficient enough for large examples. It also does not clearly scale to the first-order case.

XXX something optimistic? about trace splitting?

References

[1] Henry DeYoung and Carsten Schürmann. Linear logic voting protocols. In *VoteID 2011*, pages 53–70. Springer LNCS 7187, 2012.

[2] Max I Kanovich. Petri nets, horn programs, linear logic and vector games. *Annals of Pure and Applied Logic*, 75(1):107–135, 1995.

[3] Richard M Karp and Raymond E Miller. Parallel program schemata. *Journal of Computer and system Sciences*, 3(2):147–195, 1969.

[4] Jérôme Leroux. The general vector addition system reachability problem by presburger inductive invariants. *Logical Methods in Computer Science*, 6(3), 2010.

[5] Jérôme Leroux. Presburger vector addition systems. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 23–32, 2013.

[6] George S. Sacerdote and Richard L. Tenney. The decidability of the reachability problem for vector addition systems (preliminary version). In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, STOC '77, pages 61–76, New York, NY, USA, 1977. ACM.

[7] Robert J. Simmons. *Substructural Logical Specifications*. PhD thesis, Carnegie Mellon University, 2012.