Chapter 6

Reasoning About Linear Logic Programs

6.1 Introduction

Specifications in linear logic involve complex, interdependent state transformations that are not always easy to reason about, especially as programs grow. Many programs implicitly encode *invariants* that the programmer holds in their head while adding complexity. For instance, in the interactive fiction example described Section 4.6.3, the programmer must take great care not to write any rules that discard an atom of the form `at player Room`. The whole program's correctness hinges on the implict assumption that the player is always `at` some location in the world, and we can very easily create programs that break such assumptions. When the language permits writing such fragile code, we cannot make a strong claim about its usability, due to the potential for novices to make errors like these without the ability to know what they did wrong.

On the other end of a spectrum of expressive power for constraints, we can also consider full verification of high-level properties of a game such as solvability. Such constraints have been used in generative methods [SM11], including the automatic generation of playable games. Extending invariant checking to these domains would mean not only producing well-formed starting configurations (such as level configuration) but also proving that everything the player can *do* in a generated game or level does not break its well-formedness.

To these ends, we propose constructing automated reasoning tools for linear logic programs that enable the explicit statement and automatic checking of program invariants. In this chapter, we present partial progress toward this goal and discuss challenges to solving it in full generality. We present three candidate logics for stating invariants, several manual proofs illustrating each approach, and a decidability result for automatically verifying a fragment of the language.

6.1.1 Example: Blocks World

Consider the following specification of the "blocks world" domain from AI planning [Nil14]: the domain includes blocks stacked on a table and a robotic arm that can pick up blocks

and set them down on top of one another or on the table.

```
block : type.
on block block : pred.    % one block on top of another.
on_table block : pred.    % the block is on the table.
clear block : pred.       % the block has nothing on it.
arm_holding block : pred. % robot arm holds the block.
arm_free : pred.          % robot arm holds nothing.

% pick up a block from the table.
pickup_from_table : on_table B * clear B * arm_free -o arm_holding B.

% put down a block on the table.
putdown_on_table  : arm_holding B -o on_table B * clear B * arm_free.

% pick up a block from atop another block.
pickup_from_block : on BHi BLow * clear BHi * arm_free
                      -o clear BLow * arm_holding BHi.

% put down a block on top of another block.
putdown_on_block  : clear BLow * arm_holding BHi
                      -o on BHi BLow * clear BHi * arm_free.
```

To fully specify the domain, we state some assumptions about well-formed states and interactions. For instance, we assume that blocks are arranged in linear stacks with the table at the bottom, we assume that the arm always holds at most one block, and we assume that there is a unique top block of every stack that is marked as "clear." In order for us to verify all of these assumptions, we need to know that they hold of any initial configuration of blocks, and we need to know that the rules in the program *preserve* the same assumptions. If this were a terminating program rather than a potentially-infinite simulation, we might also want to state and prove properties of its execution in terms of what *final* configurations look like, possibly as a function of initial configurations.

The hardest problem in the collection of problems just described seems to be *invariant preservation*, i.e. a property of a program that says its rules preserve some constraint on the shape of the context. The problem bears some similarity to that of specifying well-formedness constraints on LF contexts, which are expressed with *regular worlds* in the Twelf implementation [PS98], in that we want to specify general patterns or schema for predicates that must appear alongside one another pertaining to specific terms. Ideally, checking invariant preservation should not depend on a *domain instance*. An example of a domain instance for blocks world is a specification of the particular blocks and starting configuration in a given simulation, such as the following:

```
a : block.
b : block.
c : block.

context init =
{ on_table a, clear a,
```

```
on_table b, on c b, clear c,
arm_free}.
```

This part of the program can be thought of as *input* to the rule set, in the sense that the rules should make sense *for any* well-formed set of blocks and initial configuration. Thus a checking algorithm for invariant preservation ought not to depend on this part of the specification. [1]

To check initial states, final states, and invariants, we need a way of describing context schema (sets of configurations representing a property of interest) and a framework in which to express and prove the program's relationship to those schema. This chapter describes three candidate logics for stating and proving these properties: *meta-linear logic*, *generative signatures*, and *consumptive signatures*. We show examples of proofs carried out in each of these logics for domains and properties of interest to this thesis. We describe the limitations and potential for each approach's automatability, and for the case of generative signatures, we describe a decidability result for the propositional case of Ceptre. We conclude by suggesting avenues for further investigation of this still-open problem.

6.2 Meta-Linear Logic

The well-formedness conditions of the Blocks World domain can be decomposed into two constraints on blocks and one on the robot arm, which may be informally stated as follows:

1. The first constraint says a block must have a well-formed base: either it is on the table, it is on top of another block, or it is held by the robot arm.
2. The second constraint says a block must have a well-formed top: either it is clear, there is another block on top of it, or it is held by the robot arm.
3. The third constraint says that the robot arm must either be free or holding a block.

Note that each of these descriptions independently seem to describe some mutual exclusion between resources, which could be described through linear logic's additive disjunction operator $\oplus$. These properties do not *partition* the context, however. For instance, the robot arm holding a block can satisfy all three of them, but we do not want to suggest that there should be three robot arm-related resources, when in fact we intend there to be exactly one.

We can describe these possibly-overlapping properties independently in terms of *restrictions* of the context Δ to certain predicate sets related to certain entities (usually predicate indices) over which properties are described, written $\Delta\lfloor_{\{p_1,\ldots,p_n\}}$ where p_i are predicate patterns (predicates where each index either refers to a variable bound by a quantifier or a wildcard _ satisfied by any index). We can also quantify over these entities in the formula, universally at the outside and existentially on the inside. For instance, the three properties above map onto the following formal statements defining

[1] However, see Section 6.6.1 for an example of a specification that might break this assumption.

what it means for an argument context Δ to satisfy the invariant (abbreviating `on_table` to ot, `arm_holding` to ah, `arm_free` to af, and `clear` to cl):

- Invariant BW_1 (block bottom well-formedness):

$$\forall b : \mathsf{block}.\Delta\downarrow_{\{\mathsf{ot}\ b,\ \mathsf{on}\ b\ \_,\ \mathsf{ah}\ b\}} \vDash (\mathsf{ot}\ b) \oplus (\exists b'.\mathsf{on}\ b\ b') \oplus (\mathsf{ah}\ b)$$

- Invariant BW_2 (block top well-formedness):

$$\forall b : \mathsf{block}.\Delta\downarrow_{\{\mathsf{cl}\ b,\ \mathsf{on}\ \_\ v,\ \mathsf{ah}\ b\}} \vDash (\mathsf{cl}\ b) \oplus (\exists b'.\mathsf{on}\ b'\ b) \oplus (\mathsf{ah}\ b)$$

- Invariant BW_3 (arm well-formedness):

$$\Delta\downarrow_{\{\mathsf{ah}\ \_,\ \mathsf{af}\ \}} \vDash (\exists b.\mathsf{ah}\ b) \oplus \mathsf{af}$$

Once a specific ground term index is plugged into this formula and the restriction is calculated, satisfiability $\vDash$ is determined simply by producing a derivation in linear logic.

To check a *program rule*, we must parameterize over frame contexts in which the rule might apply. For example, consider the "pick up from table" rule, which has the form

```
ontable B * clear B * arm_free -o arm_holding B
```

In order for this rule to apply to a context Δ, that context must have the form

$$\Delta',\ \mathsf{ontable}\ b,\ \mathsf{clear}\ b,\ \mathsf{arm\_free}$$

(where b is a ground term), and the state after application of the rule will be

$$\Delta',\ \mathsf{arm\_holding}\ \ b$$

We quantify universally over the context Δ', which must be reasoned about parametrically. The invariant itself quantifies over block terms B, and to show that it holds of the resulting state, we must reason parametrically over those as well. The *locality* or frame property of linear logic transitions means that whichever ones might appear in Δ' but not in the rule will remain unchanged and easy to reason about. That leaves reasoning about the indices that do appear in the rule with respect to the invariant. [2]

An invariant I is preserved by a given rule taking a state Δ to a state Δ' exactly when $I(\Delta)$ implies $I(\Delta')$. We can show that each of the three parts of the blocks world invariant described above apply to the `pickup_from_table` rule as follows.

Proposition 6.2.1 (Rule preserves blocks world invariants)**.** *Given any* Δ'*,*
let $\Delta = \Delta',\ \mathsf{ontable}\ b,\ \mathsf{clear}\ b, \mathsf{arm\_free}$ *.*
For each $I \in \{BW_1, BW_2, BW_3\}$*, if* $I(\Delta)$ *then* $I(\Delta',\ \mathsf{arm\_holding}\ b)$*.*

[2] This convenience depends crucially on the "well-moded" nature of rules: any variables mentioned in the consequent of the rule must relate somehow to variables in the antecedent. This means that even with infinite term domains, we never need to reason inductively over all inhabitants.

Proof. **Invariant BW_1: `pickup_from_table` preserves block bottom well-formedness** Intuitively, this property holds because the only block affected is the one manipulated by the rule, and its bottom's initial ontable property is simply replaced by an equally satisfactory arm_holding property. The proof follows.

By assumption (BW_1 holds of the input context),

$$\forall b'{:}\mathsf{block}.(\Delta',\ \mathsf{ontable}\ b,\ \mathsf{clear}\ b, \mathsf{arm\_free}\ )\lfloor_{\{\mathsf{ot}\ b', \mathsf{on}\ b'\ \_, \mathsf{ah}\ b'\}} \vDash (\mathsf{ot}\ b') \oplus (\exists b''.\mathsf{on}\ b'\ b'') \oplus (\mathsf{ah}\ b')$$

Let S be the restriction set given above, $\{\mathsf{ot}\ b', \mathsf{on}\ b'\ \_, \mathsf{ah}\ b'\}$.

What we need to show is that for all blocks c,

$$(\Delta',\ \mathsf{arm\_holding}\ b)\lfloor_{\{\mathsf{ot}\ c, \mathsf{on}\ c\ \_, \mathsf{ah}\ c\}} \vDash (\mathsf{ot}\ c) \oplus (\exists b''.\mathsf{on}\ c\ b'') \oplus (\mathsf{ah}\ c)$$

Assume an arbitrary block c for which to prove this fact.

There are two cases: $c = b$ (the block referred to in the particular transition) or $c \neq b$. In the case where b and c are distinct, $\Delta\lfloor_S = \Delta'\lfloor_S$, and also $(\Delta',\ \mathsf{arm\_holding}\ b)\lfloor_S = \Delta'\lfloor_S$. Then, what we know and need to show are the same, so the case is satisfied.

In the case where $b = c$, then we know $\Delta\lfloor_S = \Delta'\lfloor_S, \mathsf{on\_table}\ b$.

By inversion, the proof that $\Delta'\lfloor_S, \mathsf{on\_table}\ b \vDash \mathsf{ontable}\ b \vdash (\mathsf{ot}\ b) \oplus (\exists b'.\mathsf{on}\ b\ b') \oplus (\mathsf{ah}\ b)$ can only take the following form (according to the semantic interpretation of satisfiability given in 6.2.2, which essentially reduces to linear sequent provability):

$$\frac{\dfrac{}{\mathsf{ontable}\ b \vdash \mathsf{ontable}\ b}\ \mathsf{init}}{\mathsf{ontable}\ b \vdash (\mathsf{ot}\ b) \oplus (\exists b'.\mathsf{on}\ b\ b') \oplus (\mathsf{ah}\ b)}\ \oplus R_1$$

Thus, $\Delta'\lfloor_S$ must be empty.

Now we need to show

$$(\Delta',\ \mathsf{arm\_holding}\ b)\lfloor_S \vDash (\mathsf{ot}\ b) \oplus (\exists b'.\mathsf{on}\ b\ b') \oplus (\mathsf{ah}\ b)$$

Since $\Delta'\ \lfloor_S$ is empty,

$$\begin{aligned}(\Delta',\ \mathsf{arm\_holding}\ b)\lfloor_S &= \Delta'\lfloor_S, \mathsf{arm\_holding}\ b \\ &= \mathsf{arm\_holding}\ b\end{aligned}$$

So we equivalently need to show $\mathsf{arm\_holding}\ b \vDash (\mathsf{ot}\ b) \oplus (\exists b'.\mathsf{on}\ b\ b') \oplus (\mathsf{ah}\ b)$ which holds by derivation from $\oplus R$ rules. □

Invariant BW_2: The rule `pickup_from_table` preserves block top well-formedness

Intutitively, this property holds because the only block affected is the one manipulated by the rule (B), and its bottom's initial ontable property is simply replaced by an equally satisfactory arm_holding property. The proof follows.

By assumption and expansion of BW_1, for all $b : block$, and in particular for the block b specified by the transition, $\Delta\lfloor_{\{\mathsf{ot}\ b,\ \mathsf{on}\ b\ \_,\ \mathsf{ah}\ b\}} \vDash (\mathsf{ot}\ b) \oplus (\exists b'.\mathsf{on}\ b\ b') \oplus (\mathsf{ah}\ b)$. Intuitively, this property holds because the only block affected is the one manipulated by the rule, and its top's initial clear property is simply replaced by an equally satisfactory arm_holding property. The proof follows.

Let b be the block index chosen by the transition. Let the restriction set S be $\{\mathsf{cl}\ b, \mathsf{ah}\ b, \mathsf{on}\ \_\ b\}$.

We know by assumption that

$$\Delta\lfloor_S \vDash (\mathsf{cl}\ b) \oplus (\mathsf{ah}\ b) \oplus (\exists b'.\mathsf{on}\ b'\ b)$$

and since $\mathsf{clear}\ b$ satisfies the disjunction, $\Delta'\lfloor_S$ is empty by the same inversion reasoning as previously.

We need to show $(\Delta', \mathsf{ah}\ b)\lfloor_S \vDash (\mathsf{cl}\ b) \oplus (\mathsf{ah}\ b) \oplus (\exists b'.\mathsf{on}\ b'\ b)$.

Since $\Delta'\lfloor_S$ is empty, it suffices to show $\mathsf{ah}\ b \vdash (\mathsf{cl}\ b) \oplus (\mathsf{ah}\ b) \oplus (\exists b'.\mathsf{on}\ b'\ b)$, which is derivable by rules. □

Proof for BW_3 (`pickup_from_table` preserves arm well-formedness):
Intuitively, this property holds because the arm's initial $\mathsf{arm\_free}$ property is replaced by an equally satisfactory $\mathsf{arm\_holding}$ property. The proof follows.

Let S be the restriction set $\{\mathsf{af}, \mathsf{ah}\ \_\}$.

We know by assumption that

$$\Delta\lfloor_S \vDash (\exists b.\mathsf{ah}\ b) \oplus \mathsf{af}$$

and since $\mathsf{arm\_free}$ satisfies the disjunction, $\Delta'\lfloor_S$ is empty by the same inversion reasoning as previously.

We need to show $(\Delta', \mathsf{ah}\ )\lfloor_S \vDash (\exists b.\mathsf{ah}\ b) \oplus \mathsf{af}$

Since $\Delta'\lfloor_S$ is empty, it suffices to show $\mathsf{ah}\ b \vdash (\exists b.\mathsf{ah}\ b) \oplus \mathsf{af}$, which is derivable by rules. □

So far we have only proven the invariant for one rule, intentionally chosen as one of the simpler ones in the program. In order to give a representative sample of this method, we now show consider a rule manipulating the *interaction* between two blocks, which we also want to preserve the invariants. Below, we give the intuition for the proof of invariant preservation for the `putdown_on_block` rule. There is one case that differs in format from the cases in the previous proof: in particular, invariant BW_1 for block b', the preservation of the bottom block's bottom well-formedness.

Proposition 6.2.2. *For any Δ', let $\Delta = \Delta',\ \mathsf{ah}\ b,\ \mathsf{cl}\ b'$. For each $I \in \{BW_1, BW_2, BW_3\}$, if $I(\Delta)$ then $I(\Delta',\ \mathsf{on}\ b\ b',\ \mathsf{cl}\ b,\ \mathsf{af}\ )$.*

Proof Sketch: We need to reason about each of the two blocks mentioned in this rule, b and b', and ensure that BW_1 and BW_2 holds for each of them.

Invariant BW_1 (bottom well-formedness) for block b:
This property holds because $\mathsf{ah}\ b$ is replaced by $\mathsf{on}\ b\ b'$.

Invariant BW_2 (top well-formedness) for block b:
This property holds because $\mathsf{ah}\ b$ is replaced by $\mathsf{clear}\ b$.

Invariant BW_1 (bottom well-formedness) for block b':
This property holds because the rule does not manipulate this property, i.e. predicates concerning it are not mentioned to the right nor the left of the rule.

Invariant BW_2 (top well-formedness) for block b':
This property holds because $\mathsf{clear}\ b'$ is replaced by $\mathsf{on}\ b\ b'$.

Invariant BW_3 (arm well-formedness)
This property holds because $\mathsf{ah}\ b$ is replaced by af .

6.2.1 Example: Tower of Hanoi

We continue to illustrate this meta-logical approach by example, this time with a variation on the blocks world domain that is closer to a games application: Tower of Hanoi. This example also involves the persistent, backward-chaining component of the language and allows us to illustrate our treatment of such constructs in program rules.

Define the Tower of Hanoi domain as follows: there is a finite number of posts on which rings can be placed. A robot arm may either be holding a ring or free. A ring R may only be placed on top of another ring R' if R is smaller than R'. To make our encoding more concise, we introduce a general `place` type to represent either an empty post or the top of a ring. The code follows:

```
ring : type.
smaller ring ring : bwd.

place : type.
post : type.
top_of ring : place.
bottom post : place.

on ring place : pred.
clear place : pred.
arm_free : pred.
arm_holding ring : pred.

pickup : clear (top_of R) * on R P * arm_free
      -o arm_holding R * clear P.

putdown_on_ring : arm_holding R * clear (top_of R') * smaller R R'
      -o arm_free * on R (top_of R') * clear (top_of R).

putdown_on_post : arm_holding R * clear (bottom P)
      -o arm_free * on R (bottom P) * clear (top_of R).
```

An example instantiation of this domain is:

```
p1 : post.  p2 : post.  p3 : post.
r1 : ring.  r2 : ring.  r3 : ring.

smaller r1 r2.
smaller r1 r3.
smaller r2 r3.

context init =
{clear (bottom p2), clear (bottom p3),
    on r3 (bottom p1), on r2 (top_of r3), on r1 (top_of r2),
    clear (top_of r1), arm_free}.
```

Runnable Ceptre code for this example can be found in Appendix C.3.

The invariant we want to show preserved is that whenever a ring is on another ring, the top ring is smaller. Here is a candidate statement of this invariant in the meta-logic:

$$I_{ToH^*}(\Delta) = \forall r, r'.\Delta\lfloor_{\{\mathsf{on}\ r\ (\mathsf{top\_of}\ r')\}} \vDash ((\mathsf{on}\ r\ (\mathsf{top\_of}\ r'))\otimes !\mathsf{smaller}\ r\ r') \oplus 1$$

This invariant says that any two rings in the program are either in the `on` relation, in which case they are accompanied by a provable `smaller` relation between them, or there is no such relationship between them.[3]

In order to make our proofs go through, however, we will need to strengthen the invariant to include one of the original properties of the blocks world program: in particular, the fact that rings have mutually exclusive predicates referring to what is under them and what is atop them. It suffices to supply just one of these properties, e.g.:

$$\begin{aligned} I_{ToH}(\Delta) = \\ \forall r.\Delta\lfloor_{\{\mathsf{on}\ r\ \_,\ \mathsf{ah}\ r\}} \vDash\ & (\exists r'.(\mathsf{on}\ r\ (\mathsf{top\_of}\ r'))\otimes !\mathsf{smaller}\ r\ r') \\ & \oplus (\exists p.(\mathsf{on}\ r\ (\mathsf{bottom}\ p))) \\ & \oplus (\mathsf{arm\_holding}\ r) \end{aligned}$$

The proof that this invariant holds of the program is given rule-by-rule. For rule `pickup`:

Proposition 6.2.3 (Rule preserves Tower of Hanoi invariant)**.** *For all* Δ', r : ring, *and* p : place, *if* $I_{ToH}(\Delta', \mathsf{clear}\ (\mathsf{top\_of}\ r), \mathsf{on}\ r\ p, \mathsf{arm\_free}\ )$, *then* $I_{ToH}(\Delta', \mathsf{arm\_holding}\ r, \mathsf{clear}\ p)$.

Proof. Let $\Delta = \Delta', \mathsf{clear}\ (\mathsf{top\_of}\ r), \mathsf{on}\ r\ p, \mathsf{arm\_free}$. For most instantiations of the restriction set, the proof will be trivial, since the rule does not manipulate any instances of the restriction set unless the argument is r. In that case, let S be the restriction set $\{\mathsf{on}\ r\ \_, \mathsf{ah}\ r\}$, and let A be the formula

$$\begin{aligned} & (\exists r'.(\mathsf{on}\ r\ (\mathsf{top\_of}\ r'))\otimes !\mathsf{smaller}\ r\ r') \\ & \oplus (\exists p'.(\mathsf{on}\ r\ (\mathsf{bottom}\ p'))) \\ & \oplus (\mathsf{arm\_holding}\ r) \end{aligned}$$

By assumption, $\Delta\lfloor_S \vDash A$. The restriction $\Delta\lfloor_S$ computes to $\Delta'\lfloor_S, \mathsf{on}\ r\ p$, and since $\mathsf{on}\ r\ p$ suffices to prove the disjunction (either the first or second disjunct), $\Delta'\lfloor_S = \cdot$.

Need to show: $(\Delta', \mathsf{ah}\ r, \mathsf{cl}\ p)\lfloor_S \vDash A$. $(\Delta', \mathsf{ah}\ r, \mathsf{cl}\ p)\lfloor_S = \mathsf{ah}\ r$ by reduction. $\mathsf{ah}\ r \vdash A$ by rules (the single assumption satisfies the right disjunct).

□

[3] This invariant may seem awkward compared to a similar implication-flavored statement: $\mathsf{on}\ r\ (\mathsf{top\_of}\ r') \supset !\mathsf{smaller}\ r\ r'$. However, this implication sits at the meta-logical level, and we do not wish to include such a connective in our restricted metalogic for the sake of potential automation. On the other hand, in classical logic we have the equivalence $A \supset B \equiv (\neg A) \vee B$, which gives us a *positive* (in the focusing sense; see Chapter 2) characterization of implication more suitable to automatic search. This invariant statement is essentially a reflection of this equivalence, which is valid in this logic because we consider the presence or absence of an atom in the context to be decidable—i.e. in a context restricted to the set $\{a\}$, the proposition $a^k \oplus 1$ will always hold for some k, which is analogous to the law of excluded middle in classical logic.

For rule `putdown_on_ring`:[4]

Proposition 6.2.4. *For all* Δ' *and* r : ring, *if* $I_{ToH}(\Delta',\ \mathsf{arm\_holding}\ r, \mathsf{clear}\ (\mathsf{top\_of}\ r'))$ *and* $\Sigma \vdash \mathsf{smaller}\ r\ r'$, *then* $I_{ToH}(\Delta',\ \mathsf{arm\_free}\ ,\ \mathsf{on}\ r\ (\mathsf{top\_of}\ r'),\ \mathsf{clear}\ (\mathsf{top\_of}\ r))$.

Proof. Let $\Delta = \Delta',\ \mathsf{arm\_holding}\ r, \mathsf{clear}\ (\mathsf{top\_of}\ r')$. The proof is trivial for all instantiations of the invariant except in cases where the restriction set mentions r, so let us consider the instantiation of the invariant at r for which the proof is nontrivial. Let S be the restriction set $\{\mathsf{on}\ r\ \_,\ \mathsf{ah}\ r\}$. Let A be the formula

$$\begin{aligned}&(\exists r'.(\mathsf{on}\ r\ (\mathsf{top\_of}\ r'))\otimes !\mathsf{smaller}\ r\ r')\\ &\oplus (\exists p'.(\mathsf{on}\ r\ (\mathsf{bottom}\ p')))\\ &\oplus (\mathsf{arm\_holding}\ r)\end{aligned}$$

We know $\Delta\lfloor_S \vDash A$, and in particular the premise $\mathsf{ah}\ r$ satisfies the third disjunct, so $\Delta'\lfloor_S$ is constrained to be empty. We need to show

$$(\Delta', \mathsf{af}\ , \mathsf{on}\ r\ (\mathsf{top\_of}\ r'),\ \mathsf{clear}\ (\mathsf{top\_of}\ r))\ \lfloor_S \vDash A$$

The context restriction computes to $\mathsf{on}\ r\ (\mathsf{top\_of}\ r')$ which together with the assumption $!\mathsf{smaller}\ r\ r'$ satifies the first disjunct. □

For rule `putdown_on_post`:

Proposition 6.2.5. *For all* Δ', r : ring, *and* p : post, *if* $I_{ToH}(\Delta',\ \mathsf{clear}\ (\mathsf{bottom}\ p),\ \mathsf{arm\_holding}\ r)$, *then* $I_{ToH}(\Delta',\ \mathsf{arm\_free}\ ,\ \mathsf{on}\ r\ (\mathsf{bottom}\ p),\ \mathsf{clear}\ (\mathsf{top\_of}\ r))$.

Proof. Let $\Delta = \Delta',\ \mathsf{arm\_holding}\ r,\ \mathsf{clear}\ (\mathsf{bottom}\ p)$. The invariant holds of Δ instantiated at r because of the $\mathsf{arm\_holding}\ r$ premise, which is straightforwardly replaced by the $\mathsf{on}\ r\ (\mathsf{bottom}\ p)$ consequent. This proof follows the same pattern we have now seen several times. □

6.2.2 Potential for Automation

In order to discuss potential algorithms for deciding a fragment of this meta-logic, we need to define that fragment. To do so, we constrain the grammar of the meta-logic to formulas ϕ of the following form:

$$\begin{aligned}\phi &::= \lambda\delta.\gamma\\ \gamma &::= \forall x{:}\tau.\gamma \mid \Delta\lfloor_S \vDash \psi\\ \psi &::= a \mid \psi\otimes\psi \mid \psi\oplus\psi \mid \exists x{:}\tau.\psi \mid 1\\ S &::= \cdot \mid \mathsf{pat}, S\\ \mathsf{pat} &::= a\ \mathsf{tpat}_1 \ldots \mathsf{tpat}_n\\ \mathsf{tpat} &::= \_ \mid t \mid c\ \mathsf{tpat}_1 \ldots \mathsf{tpat}_n\end{aligned}$$

[4]Note: since this rule involves a persistent premise, we must explicitly mention the signature Σ supplying the backward-chaining rules defining persistent propositions, which we had previously been leaving silent. This treatment emerges from what it means for a transition to apply; see Chapter 3 for details.

The grammar for terms t, term constants c, and atomic predicates a is borrowed from Ceptre's grammar, given in Chapter 4.

Then we give an interpretation of each component when a formula ϕ is applied to a context Δ (with respect to some type header Σ):

$$\begin{array}{rcl}
[\![\forall x{:}\tau.\gamma]\!] & = & \Sigma \vdash t : \tau \text{ implies } \Delta \vDash [t/x]\gamma \\
[\![\Delta \vDash a]\!] & = & \Delta = \{a\} \\
[\![\Delta \vDash \psi_1 \otimes \psi_2]\!] & = & \Delta = \Delta_1, \Delta_2 \text{ and } \Delta_1 \vDash \psi_1 \text{ and } \Delta_2 \vDash \psi_2 \\
[\![\Delta \vDash \psi_1 \oplus \psi_2]\!] & = & \Delta \vDash \psi_1 \text{ or } \Delta \vDash \psi_2 \\
[\![\Delta \vDash \exists x{:}\tau.\psi]\!] & = & \text{there exists } t \text{ s.t. } \Sigma \vdash t : \tau \text{ and } \Delta \vDash [t/x]\psi \\
[\![\Delta \vDash 1]\!] & = & \Delta = \cdot
\end{array}$$

Pattern restriction $\Delta\lfloor_S$ computes another context Δ', defined approximately as the intersection of Δ and S where any wildcard patterns _ in S are considered equivalent to any term. We write that an atom p *matches* a pattern pat with the notation $p \triangleright \mathsf{pat}$, which is defined next.

Pattern restriction:

$$\begin{array}{rcl}
\cdot\lfloor_S & = & \cdot \\
(\Delta, p)\lfloor_S & = & \Delta\lfloor_S, p \quad \text{if } p \in^* S \\
 & = & \Delta\lfloor_S \quad \text{otherwise} \\
p \in^* S & \text{iff} & \exists \mathsf{pat} \in S. p \triangleright \mathsf{pat}
\end{array}$$

Pattern matching:

$$\begin{array}{rcl}
t & \triangleright & t \\
t & \triangleright & _ \\
c\ t_1 \ldots t_n & \triangleright & c\ \mathsf{tpat}_1 \ldots \mathsf{tpat}_n \text{ iff } t_1 \triangleright \mathsf{tpat}_1 \text{ and } \ldots \text{ and } t_n \triangleright \mathsf{tpat}_n \\
a\ t_1 \ldots t_n & \triangleright & a\ \mathsf{tpat}_1 \ldots \mathsf{tpat}_n \text{ iff } t_1 \triangleright \mathsf{tpat}_1 \text{ and } \ldots \text{ and } t_n \triangleright \mathsf{tpat}_n
\end{array}$$

Checking that a given context matches a formula in this fragment of the logic for a given context has a straightforward correspondence to the positive fragment of first-order linear logic, which is a subset of MALL and thus decidable. Including the exponential operator ! in the limited fashion demonstrated by the Tower of Hanoi example, i.e. to express constraints on indices whose provability can be fully extricated from the linear components, does not seem to affect this decidability in any obvious way, but we currently leave its decidability as conjecture.

For invariant checking, i.e. checking that a given transition $\Delta \to \Delta'$ enabled by the program under scrutiny preserves a property given in this language, we would need to codify the proof technique embodied by the manual proofs given above in such a way that were guaranteed to terminate, including inversion of assumed derivations.

Here is a sketch of the algorithm, generalized from the previous examples:

To check an invariant $\phi = \lambda\delta.\forall x{:}\vec{\tau}.\delta\lfloor_S \vDash \psi$ of a transition $\Delta \to \Delta'$:

1. Expand $\phi(\Delta)$. Generate fresh symbols for each $\forall$-quantified variable x_i in $\phi(\Delta')$.
2. Enumerate cases for terms in Δ being equal or not equal to those symbols.
3. For each case, compute the restriction set S and determine possible inversions based on the rules for decomposing the inner formula ψ (analogous to linear sequent calculus right rules). These introduce equality constraints on the restriction set of the input context Δ and on the part it shares with Δ'.
4. Using those constraints and, again, decomposition rules for the invariant formula, determine satisfiability of $\Delta' \vDash \psi$.

This process does not straightforwardly correspond to a known-decidable procedure, but since the satisfiability semantics obey a subformula property in the same sense as their corresponding linear logic rules, we expect that inversion is a computable process. We also have not yet encountered examples in this fragment that require inductive lemmas. Thus, we conjecture that checking preservation of invariants described in this meta-logic is decidable. It remains to prove that this is the case, as well as to prove soundness and completeness with the logical derivability-based notion of invariant preservation. We leave further formalization and proof to future work.

6.2.3 Limitation: Recursive Predicates

Working in a restricted meta-logic to specify program invariants has some drawbacks. For instance, some specifications of well-formed states are simply inaccessible. Consider an encoding of linked lists in linear logic wherein memory areas are encoded as abstract *destinations* (terminology borrowed from the idea of "destination-passing style" [CPWW03]) used as indices to a predicate describing data at a location as well as a reference to the next location:

```
data : type.
location : type.
node location data location : pred.
```

The list `[a,b,c]` would be encoded as the domain instance:

```
a : data. b : data. c : data.
l1 : location. l2 : location. l3 : location.
end : location.
context list_abc = {node l1 a l2, node l2 b l3, node l3 c end}.
```

We can write programs over this data that, for instance, delete a node from the list (at location `L'`):

```
delete : at L V L' * at L' _ L'' -o at L V L''.
```

A common verification problem for reasoning about programs at a memory layout level is *shape analysis* [SRW02, DOY06]: can we reason that a given segment of memory has a particular shape (for instance binary tree or linked list) and that a program that manipulates it, possibly dynamically allocating memory, preserves that shape? Previously, linear logic has been investigated as a candidate for expressing these program

constraints in terms more abstract than memory locations [JW06], and it seems that such analyses should also extend to linear logic programs themselves.

In particular, for this example, we might like to state that the linear, non-circular, linked list structure of our memory layout encoding is preserved by the deletion rule. But such a property cannot be given by local characterization of a single index; it must refer to the structure of how indices are shared between predicates in the context. We also cannot give it as a persistent property defined outside the state evolution rules, since it refers to facts (like the way nodes are linked) that fundamentally change during execution.

Instead, such a property is more naturally given recursively, by saying either the list is empty, or contains a node pointing to another well-formed linked list. One candidate approach for stating this kind of invariant is to extend the meta-logic with a notion of recursive predicate. Such extensions to first-order linear logic have been explored and proven sound [Bae08]. However, this extension may complicate decidability of the approach, and does not offer clear means for automation. We describe a related approach more fully as part of a different technique in Section 6.3.

6.3 Consumptive Invariants

Aside from automation concerns, we also hope to extend the expressiveness of the meta-logic by allowing the description of recursive invariants, such as those described in the shape analysis literature. We describe a technique that can *dynamically* check such invariants next.

This technique hinges on the observation that descriptors of context properties resemble grammars that themselves can be expressed in linear logic. And we can write recursive linear logic programs to express those properties via backward-chaining.

6.3.1 Backward-Chaining Linear Logic Programs

Linear logic programs may be given a backward-chaining interpretation so long as each clause only has a single, atomic consequent, i.e. takes the form $A \multimap p$. In concrete syntax, instead of `p1 * ... * pn -o p` for such a rule we will write the backwards, curried form `p o- p1 o- ... pn`, mimicking the syntax for persistent backward chaining programs.

We refer to p as the *head* of such a clause, and proof search may be carried out on a particular atom by matching it against the heads of all logic program clauses, then matching their premises as new goals, backtracking when search along a particular branch fails, just like persistent backward chaining. But linear rules still have a resource-based meaning, i.e. for a rule `p o- p1 o- p2` to succeed at establishing `p` in a context Δ means that Δ must be partitionable into Δ_1, Δ_2 such that `p1` consumes all of Δ_1 and `p2` consumes all of Δ_2.

6.3.2 Example: Linked List Shape Analysis

Consider the linked list example given in Section 6.2.3. We can write a simple recursive predicate defining a well-formed linked list *segment* with two arguments, s for the beginning location of the list and e for the ending location. The linked list segment is well formed if either $s \doteq e$ (the segment is empty) or there is some node at s with "next" field l such that l and e define a well-formed segment.

This definition can be represented by the following backward-chaining linear logic program:

```
ll location location : bwd.
ll X X.
ll S E o- at S V L
       o- ll L E.
```

Simmons [Sim12] refers to this kind of specification as a *consumptive* signature, because to use it as a verification tool, we would supply a context Δ and attempt to prove $\Delta \vdash \exists s, e.\mathtt{ll}\ s\ e$, which works operationally by using the rules defining `ll` to *consume* elements of Δ until none are left.

6.3.3 Potential for Automation

By combining the restriction mechanism from the meta-linear logic approach with predicates defined recursively in linear logic, we can straightforwardly extract a *dynamic* checking algorithm: after every rule application, check that the resultant context Δ (or whatever restriction of it) satisfies the formula by simply running linear logic proof search. We have not determined to what extent such invariants can be checked of a program statically, nor to what extent the dynamic process described can be optimized not to re-compute full provability between every transition in the program.

6.3.4 Limitation: Apartness Constraints

Apart from a lack of a known static decision procedure for preservation, consumptive invariants have the following limitation: they cannot enforce that a given predicate holds only for distinct term indices. For instance, consider this candidate representation of the blocks world well-formedness property.

```
wf_bw
  o- wf_arm
  o- wf_stacks.

wf_arm o- arm_free.
wf_arm o- arm_holding X.

wf_stacks o- 1.

wf_stacks o- on_table B
```

```
          o- wf_stack B
          o- wf_stacks.

wf_stack B o- clear B.

wf_stack B o- on B' B
           o- wf_stack B'.
```

This specification does not suffice to rule out ill-formed blocks world configurations as specified in Section 6.2, because it does not enforce that a given block B is not, say, simultaneously held by the arm and on the table (or appearing in multiple stacks). The term indices of the rules may be instantiated with any substitutions, including ones that unify them with otherwise-existing terms.

6.4 Generative Invariants

A *generative* signature, in contrast to a consumptive signature, is one that describes a context property through rules that *generate* all permissible contexts, again using the analogy between logic programs and grammars. Generative signatures and their use for specifying logic program invariants were also first described in [Sim12].

Generative signatures are collections of forward-chaining rules together with a *seed* context Δ_0, usually containing a distinguished `gen` atom which is expanded by the signature. By convention, we will assume that all seed contexts take this form such that the signature itself suffices to specify the property.

At first glance, generative signatures look like consumptive signatures "with the arrows turned around:" instead of distinct `wf` (well-formedness) predicates for each portion of the context, we have distinct *generators*, analogous to nonterminals in a grammar, for each portion of the context. A complete well-formed context Δ, then, is one for which there is a transition sequence $\Delta_0 \rightarrow \Delta$ along rules given in the generative signature Σ_{gen}, where Δ contains no nonterminals. An extremely simple example is given below:

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

This signature, equipped with the seed context {`gen`}, describes contexts containing zero or more instances of `a`. Formally, the set of contexts that a signature Σ and a seed Δ_0 describes is the set of reachable contexts from Δ_0 following rules in Σ. We call such a signature and seed pair (Σ, Δ_0) a *generative invariant* of a program signature Σ' if all rules in Σ' maintain the program state's membership in the set generated by Σ seeded with Δ_0. We make this notion more precise later.

In addition to universally-quantified indices (standard logic variables), generative invariants may also include *existentially generated* variables via rules of the form `A -o exists x.B`. The existential quantifier has completely standard treatment from a forward-chaining proof search perspective; see the CLF paper [WCPW03] for a formal treatment.

6.4.1 Example: Generative Signature for Blocks World

Below we give a generative signature characterizing the blocks world domain, effectively by "turning the arrows around" in the consumptive signature. Let $\Sigma_{\mathsf{bwgen}} =$

```
gen/bw : gen -o gen_arm * gen_stacks.
gen_arm/af : gen_arm -o arm_free.
gen_arm/ah : gen_arm -o exists b. arm_holding b.
gen_stacks/done : gen_stacks -o 1.
gen_stacks/more :
  gen_stacks -o exists b. on_table b * gen_stack b * gen_stacks.
gen_stack/clear : gen_stack B -o clear B.
gen_stack/more  : gen_stack B -o exists b. on b B' * gen_stack b.
```

This signature says: in order to build a blocks world, build an arm and a set of block stacks. The arm can either be free or holding a block. A set of block stacks can be empty, or it can contain a block stack starting with a new block index, where that block is on the table, along with the rest of the set of block stacks. A block stack is indexed by its top block, and that block can either be clear, or have another block on top of it that becomes the new block stack index.

6.4.2 Generative Property Preservation

We now define what it means for program transitions to preserve generative properties.

Definition: A transition $\Delta, A \to \Delta, B$ 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 $\Delta_0 \rightsquigarrow_{\Sigma_{gen}} \Delta, B^*$.

In particular, for the blocks world example, we can show how to reason that the `pickup_from_table` rule, enabling the transition $\Delta, \mathsf{ot}\ b,\ \mathsf{af},\ \mathsf{cl}\ b \to \Delta, \mathsf{ah}\ b$, preserves the invariant by showing that:

Proposition 6.4.1. *If*

$$\{\mathsf{gen}\} \to_{\Sigma_{\mathsf{bwgen}}} \Delta, \mathsf{on\_table}\ b,\ \mathsf{arm\_free},\ \mathsf{clear}\ b$$

then

$$\{\mathsf{gen}\} \to_{\Sigma_{\mathsf{bwgen}}} \Delta, \mathsf{arm\_holding}\ b$$

Proof Sketch: The signature Σ_{bwgen} can generate $\Delta, \mathsf{on\_table}\ b, \mathsf{arm\_free}, \mathsf{clear}\ b$ in only the following way, up to concurrent equality:

1. Apply rule `gen/bw` to get context `{gen_arm, gen_stacks}`.
2. Apply rule `gen_arm/af` to get context `{arm_free, gen_stacks}`.
3. Apply some unknown rule sequence to `gen_stacks` to yield an unknown context `D1` along with another copy of `gen_stacks`.
4. Apply `gen_stacks/more` to create b along with its `on_table` property. Context is now `{arm_free, D1, on_table b, gen_stack b, gen_stacks}`.
5. Apply `gen_stack/clear` to `gen_stack b` to get `clear B`.

6. Apply some unknown rule sequence to `gen_stacks` to get a side effect nonterminal context `D2`, and eventually eliminate all nonterminals.
   The context is now `{arm_free, D1, on_table b, clear b, D2}`.
   The original frame context Δ must equal `D1, D2`.

Need to show: $\{\mathsf{gen}\} \rightarrow_{\Sigma_{\mathsf{bwgen}}} \Delta, \mathsf{arm\_holding}\ b$.

Given the proof trace above, we can modify it selectively to produce the context we need. We modify step 2 to replace `gen_arm/af` with an application of `gen_arm/ah`, which generates a block b that is indistinguishable from the specific b in question up to alpha renaming, and the predicate `arm_holding b`. We modify steps 4 and 5 to be the null transition, simply linking together the trace that generated `D1` to the trace that uses `gen_stacks` to generate `D2`, eliminating the terminals that were produced as a side-effect. □

To make the above proof sketch formal, we need to give the input trace in terms of ϵ notation (a sequence of let-bindings **let** $\langle x_1 \ldots x_n \rangle = r\ R$ witnessing the transformation $\{\mathsf{gen}\} \rightarrow_{\Sigma_{\mathsf{bwgen}}} \Delta, \mathsf{on\_table}\ b,\ \mathsf{arm\_free},\ \mathsf{clear}\ b$) and prove that it really is the only possibility, up to concurrent equality. Simmons [Sim12] carries out such proofs by proving *permutability* lemmas, which say that in any trace, we can permute the generation of the terminals we care about to the end, then reason inductively on the remaining trace prefix. The tediousness and surfeit of cases in these proofs suggest that formal argument over traces is the wrong level of abstraction for these proofs—in the case of generative well-formedness, we do not care about the *order* in which things are generated by the signature but rather only about reachability between intermediate states of the generative program. An attempt at visualizing the structure of a generative trace is given in Figure 6.1.

The ability to draw the trace this way makes several assumptions that we would need to prove about a given generative signature, such as non-interference between `gen_arm` and `gen_stacks`, and for that matter the fact that terminals cannot interfere with the trace structure. All of these assumptions suggest constraining the grammar of generative signatures themselves in such a way that their traces always have this tree-like structure, which we will do when we formalize them in an attempt to automate proof search.

6.4.3 Potential for Automation

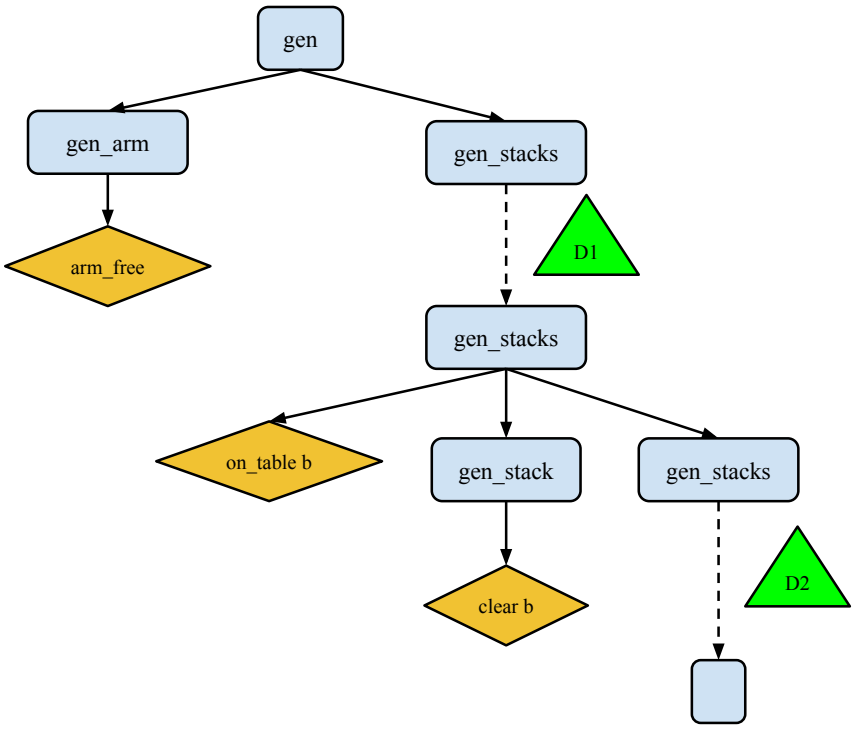
We have not yet devised a general algorithm for checking preservation of generative invariants, and we suggest that without a better layer of abstraction for reasoning about generative traces, such a task is intractable: the by-hand proofs frequently involve inductively-proven lemmas and careful scrutiny of the particular generative signature at hand. However, generative signatures so far yield the most definitive result of the three approaches: if we limit programs and generative signatures to *atomic propositions*, invariant preservation is decidable. We give a proof of this fact in Section 6.5.

Figure 6.1: Generative trace structure

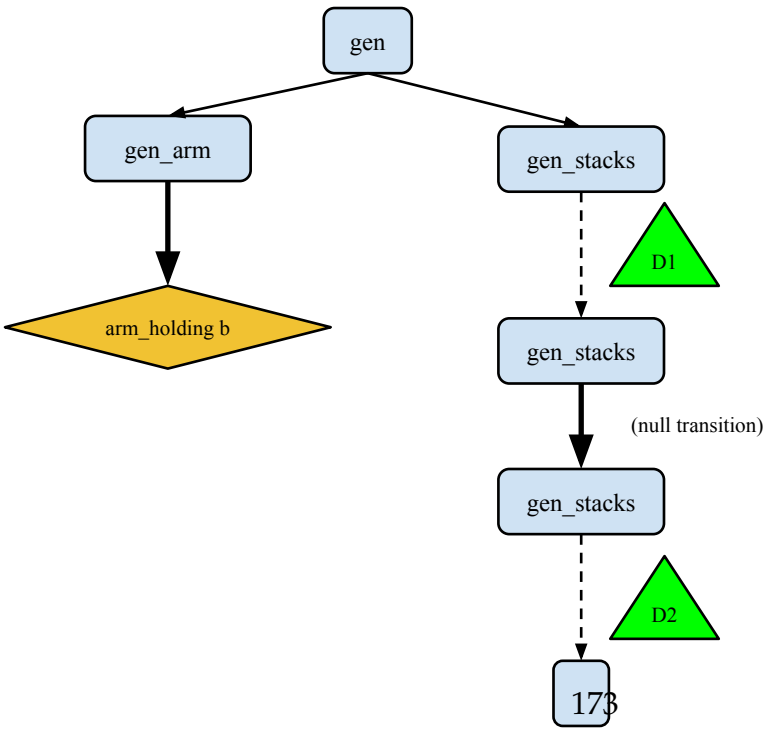
These diagrams depics the generative trace structure for blocks world well-formedness of precondition and postcondition of the rule

```
pickup_from_table : on_table B * clear B * arm_free -o arm_holding B.
```

Precondition trace:



Postcondition trace:



6.4.4 Limitations

Generative signatures resolve the major limitation of consumptive signatures, the expression of apartness constraints, through the existential quantifier, which always generates a fresh term distinct from any others previously in existence. However, this approach is not without its limitations.

For one thing, we suspect that generative signatures are a substantially less intuitive way to write specifications than, for instance, the first-order formulas of meta-linear logic discussed in Section 6.2. While meta-linear logic is a fairly direct way of translating a constraint on a program into a logical formula, generative invariants require the programmer to *operationalize* their understanding of the constraint by specifying how to generate contexts that satisfy it.

Second, there is another large class of constraints we cannot describe with generative invariants: consider describing *well-formed graphs* where edges between nodes are predicates in the program, but we do not want to create self-loops or multi-edges. However, a new node created may refer back to an old node. A candidate signature is:

```
gen/nodes : gen -o exists n:node. gen_edges n * gen.
gen_edges/edge : gen_edges N * gen_edges N'
    -o edge N N' * gen_edges N * gen_edges N'.
gen_edges/done : gen_edges N -o 1.
```

This specification has two problems: first of all, it has a rule with two nonterminals on the left, which even in the propositional case falls outside the known-decidable fragment. Second, it cannot avoid generating self-edges or multi-edges, since nothing stops `N` and `N'` from having the same node substituted for them. The existential "fresh generation" property does not help us here, because it cannot promise to keep indices apart in a non-local way, i.e. in a rule other than the one where the existential appears.

In general, there does not appear to be a way to specify formation properties in which indices can be shared arbitrarily between predicates while also enforcing certain apartness constraints. Even with meta-linear logic, we would need to introduce new apartness primitives on terms in order to handle such a specification.

6.5 Decidability of Invariant Checking for the Propositional Fragment

In this section we prove that checking whether a (propositional) generative invariant holds of a (propositional) linear logic program is decidable. We do so by modeling propositional generative signatures and programs as *vector addition systems* (first introduced by that name by Karp and Miller [KM69], although the equivalent Petri nets were devised earlier [Pet66]) 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 (despite impractical computational complexity) happens to be decidable [Sta84].

6.5.1 The Propositional Fragment

Propositional Ceptre programs are those which do not have any logic variables (Π-bound variables), and thus the available transitions at any point in the program does not depend on the term language available. Formally, it is simply a restriction of the grammar of rules to

$$\begin{aligned} A &::= S \multimap S \\ S &::= 1 \mid a \mid S \otimes S \end{aligned}$$

Instead of refering to these programs as propositional *Ceptre* programs, we will refer to them as propositional *Horn* programs by analogy with Horn clauses in standard logic programming (i.e. ones with no left-nested implication) [Kow74], which we believe is consistent with prior definitions of the Horn fragment of linear logic [Kan92, Kan95].

Below is an example of a propositional Ceptre program modeling coin exchange—`r1` exchances a dime for two nickels, and `r2` exchanges a quarter for two dimes and a nickel.

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

Even with such a limited specification, we can already ask all of the same questions we asked in the beginning:

- **What is the set of well-formed states?** In this case, it is simply any number of quarters, dimes, and nickels.
- **Which states do we take to be well-formed *initial* states?** For instance, we might specify that we start with entirely quarters, or we might allow any configuration of coins.
- **What is the set of possible states at *quiescence* (termination) of the program?** This depends on the set of initial states. If we take the initial states to be those with only quarters, then we expect to have a multiple of five nickels at the end and no other coins. If we allow any set of coins initially, then we will have an arbitrary number of nickels (and no other coins) at the end.

Indeed, the propositional fragment corresponds with Petri nets [Mur89] and has been used for quite sophisticated specifications in the games and interactive storytelling literature as well as for modeling concurrent and distributed programs. For instance, Dang et al. [DHCS11] specify a slapstick interactive storytelling scenario in which the player character is attending a party and may take various actions that have cascading effects, such as lighting a woman's cigarette:

```
light_cig : bored_woman * woman_has_cig * cig_not_lit
            -o woman_has_cig * cig_lit.
```

...which can lead to objects and animals catching fire. Because the scenes are all entirely hand-authored and specific to certain characters and objects, the whole interactive space can be described with atomic propositions.

Furthermore, first-order logic programs may be expanded to propositional ones through the process of *grounding*, if their term domains are finite. For instance, if we have the type header

```
char : type.
loc  : type.
at char loc : pred.

alice : char.
bob   : char.

den   : loc.
foyer : loc.
```

Then the rule `rule : at C L -o 1` may be expanded into the following four rules:

```
at alice den   -o 1.
at alice foyer -o 1.
at bob   den   -o 1.
at bob   foyer -o 1.
```

Such a process is typically not considered computationally practical, since the number of rules needed to represent a single rule grows combinatorially with the number of logic variables it has, and checking Presburger formulas is doubly-exponential in complexity [FR98]. However, since our only aim with this proof is to establish decidability (rather than a tractable algorithm), we note that our proof extends to these programs as well.

Thus we consider decidability of invariant checking for propositional programs to be a valuable and novel contribution to the space of validating interactive media.

6.5.2 A Tricky Example

To see why the problem of invariant preservation is a challenge, consider the following generative signature $\Sigma_{\mathtt{abc}}$.

```
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_{abc}, `gen`) describes is one where an `a` can appear alongside arbitrarily many `cs`, but a `b` requires *exactly two* `cs` alongside it.

Here are two 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`.

On the other hand, here are two 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. Rules are *context-sensitive* with respect to generative invariants; they cannot be reasoned about in isolation.

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.

6.5.3 Vector Addition Systems

In order to draw on prior work in algorithms for validating these systems, we need to understand them in terms of the formalisms chosen by that prior work. In particular, a number of useful properties have been shown of *vector addition systems*, which we employ to obtain decidability.

A *vector addition system* (VAS) V is an initial configuration $v_0 \in \mathbb{N}^n$ for a fixed dimension n, along with a set T of *transitions* describing how a configuration v may evolve. Transitions can be described as vectors $t \in \mathbb{Z}^n$ along with side conditions of the form $k \in \mathbb{N}^n$, which imposes the inequality constraint $v \geq k$. If the conditions hold of a configuration v (i.e. $v \geq k$) *and* $v + t \geq 0$, the transition t is said to be *applicable* or *fireable*, and the configuration may evolve to $v + t$.

For a transition $(t, k) \in T$ that is applicable in v, we notate a transition relation $\rightsquigarrow$, i.e. $v \rightsquigarrow v+t$, which we use in addition to its transitive closure $\rightsquigarrow^*$. The set of configurations $\{x \mid v \rightsquigarrow^* x\}$ is called the *reachability set* of v, and can be extended to sets of starting configurations as well. VASes are equivalent to Petri nets (see e.g. [Ler10, Kan95]), and for both systems, the central problem is computing reachability sets: given a set of starting configurations, what are all of the possible states the system might wind up in? This problem is known to be decidable [Pet77, ST77] (however, we do not make direct use of this fact for our proof).

The correspondence also extends to provability of a sequent in the propositional Horn fragment of linear logic. Simply put, a VAS transition rule of the form (t, k) can be interpreted as a forward-chaining 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.

In the other direction, a linear logic program with rules of the form

$$a_1^{c_1} \otimes \ldots \otimes a_n^{c_n} \multimap b_1^{d_1} \otimes \ldots \otimes b_m^{d_m}$$

(with like atoms grouped together without loss of generality) can be interpreted as a VAS by the following process:

1. Give an arbitrary canonical ordering for all atoms in the program to form the space of possible vectors. Thus each atom a corresponds with an index i_a of the vector.

2. For each rule in the above form, create a transition (t, k) where $t_i = c_i - d_i$ and $k_i = c_i$.

By way of example, all of the rules in Σ_{abc} may be translated to the following vector addition rules 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>
```

The initial configuration {gen} is represented by the vector $\langle 1, 0, 0, 0, 0\rangle$.

As mentioned previously, the decidability of this problem does not help us directly with deciding if a given program rule preserves a generative invariant. We can, however, recast the preservation problem as a VAS problem: if we want to check whether a rule $r : A \multimap B$ satisfies a generative invariant I, reformulate I as a vector addition system $V_I = (i_0, T_I)$. The rule r will correspond to some transition (t, k) (in a different vector addition system, whose full definition is not needed).

Definition: A transition (t, k) *preserves the invariant* given by VAS V_I with starting configuration i_0 iff for all configurations $v \geq k$

$$i_0 \rightsquigarrow^*_{V_I} v$$

implies

$$i_0 \rightsquigarrow^*_{V_G} v + t$$

The vector v in this definition represents an arbitrary frame context Δ, A in which the rule fires. [5]

We can make a few more definitions to simplify this characterization:

Definition: The *reachability set* of a vector v under a particular VAS V is the set of all vectors v' such that the transitive closure of all transitions in V can take v to v':

$$\mathsf{reach}_V(v) = \{v' \mid v \rightsquigarrow^*_V v'\}$$

Definition: The *postset* of a given vector set S along a transition (t, k), 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 $v' = v + t$.

Now we can reword the above criterion for preservation as simply

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

where $S = \mathsf{reach}_{V_G}(c_0)$.

[5] Note that, in general, the arity of vectors for a generative invariant will be higher than the arity for the VAS corresponding to the program we want to check—invariants contain nonterminals that do not appear in program configurations. So to make this definition more precise, the transition $i_0 \rightsquigarrow v$ should really be $i_0 \rightsquigarrow v'$ where $v'_i = v_i$ when i is within bounds for v and $v'_i = 0$ everywhere else.

In general, computing the reachability set does not help us answer this question. However, some (not all) VASes can be expressed as *Presburger Arithmetic formulas*, a subset of first-order logic propositions that may refer to multiplicaiton and addition of natural numbers (but not exponentiation). Presburger Arithmetic is known to be decidable, as shown by Presburger in 1929 (simultaneously with his presentation of the system). And if the reachability set S of a vector addition system V_I representing an invariant corresponds to a Presburger formula, then the question of whether a given rule r preserves it ($\mathsf{post}_{(r)}(S) \subseteq S$) can also be expressed as a Presburger formula. Thus, what remains is to show that the particular subset of VASes that correspond with generative invariants are *all* expressible as Presburger formulas.

6.5.4 Presburger Vector Addition Systems

In general, VASes are not expressible as Presburger Arithmetic formulas. An explanation of this fact and a characterization of the Presburger fragment is given in [Ler13]. Briefly speaking, one can encode exponentiation as a VAS, which is not within the bounds of Presburger Arithmetic. 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 VASes are exactly those which have an equivalent "flat" representation. A *flat* VAS is one that can be characterized as a finite sequence of arbitrarily-repeated finite transition sequences. More formally, consider a *word* w to be a sequence of transitions $t_1 \dots t_k$ such that t_{i+1} may always follow t_i. The Kleene closure operator on words w^* means "w repeated 0 or more times, so long as it applies." For instance, the closure t^* of a transition t corresponding to the linear logic rule $a \multimap b$ means that t may be applied as many times as there are copies of a in the configuration it starts with.

A flat language, then is one that can be written as

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

for some finite set of words $w_1 \dots w_n$.

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 [Esp97], 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. [6]

[6]Generative invariants for specifying the operational semantics of programming languages [Sim12] 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.

There exist published, proven terminating, and implemented methods to decompose VASs such as BPPs into flat languages [Fri00, FO97]. 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. We note that prior to this investigation, there was no rigorous characterization of what differentiates a *generative signature* from any other linear logic program, and a constraint like this one serves as a candidate for characterizing the space of suitable signatures.

6.5.5 Flat(able) Generative Invariants

Here is a generative invariant that is *not* flat, corresponding loosely to a propositional erasure of a well-formedness specification for blocks world.

```
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"
```

If we try to characterize the allowable traces generated by this program as a grammar, we would write something like $(g_1 g_2^* g_3)^* g_4^* g_5^*$, meaning basically: generate an arbitrary number of stacks of blocks by first generating a block on the table, then generating an arbitrary number of blocks atop it, then designating the last one as clear; finally, designate the arm as either free or holding a block. [7]

This specification is not flat because 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.

However, the set of contexts generated by this signature is equivalent to those creatable by a flat specification. In fact, the flat specification is closer to the one we have already seen:

```
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}.
```

The flat language representing the reachability set is:

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

[7] We could give a more precise characterization of the signature with additional regular language constructs, such as $(g_1 g_2^* g_3)^*(g_4|g_5|1)$, since exactly one of g_4 or g_5 will fire at the end exactly once (or neither of them will). However, prior work makes use of only the regular language constructs of concatenation and repetition, which affects their mapping into Presburger formulas. The Kleene star can be used to express this same thing because after g_4 fires, neither g_4 nor g_5 can, and if g_4 fires 0 times, then g_5 may fire 0 or 1 times.

In other words, the generative signature can be read operationally as follows: First, create an arbitrary number of seeds for new block stacks. Then, if applicable, we add blocks to all of those stacks. Then finish all of those stacks by marking their tops as clear. Then optionally generate a free arm, and if applicable (i.e. if we opted not to generate a free arm) generate an arm holding a block.

6.5.6 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, then computing an inductive definition of the union of those sets.

More precisely, to recapitulate Leroux's result [Ler13], define the *displacement* of a word $w = t_1 \dots t_k$ to be the vector $\delta(w) = \sum_{j=1}^{l} t_j$. Define also for w a configuration c_w such that $c_w(i) = \mathsf{max}\{-(t_1 + \cdots + t_j)(i) \mid 0 \leq j \leq k\}$. This value c_w denotes the minimal configuration for which there exists a run (applicable transition sequence) labeled by w.

Then, the postset of a set S under a single repeated word w is given by the following Presburger formula:

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

Then, for a VAS with starting configuration c_0 and represented as the flat language $w_1^* \dots w_n^*$, a sequence of Presburger sets C_i denoting the reachability set for the entire system is computed inductively:

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

The reachability set is given by $v \in \mathsf{post}^*(\{c_0\}) = C_n$ where n is the number of words in the flat sequence.

6.5.7 The Algorithm

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. We denote by $|\Delta_0|$, $|\Sigma|$, and so on the compilation of each of these linear logic program components into their corresponding vector addition system components.

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

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

In general, a postset $\mathsf{post}_{(t,k)}(S)$ can be denoted logically as the Presburger set $\{v \mid \exists x \in S. x \geq k \wedge x + t = v\}$.

And we can write the above subset condition as an implication; in particular, if q is the number of constants in the specification (i.e. the width of the vector):

$$\forall x, v \in \mathbb{Z}^q.(x \in S \wedge x \geq k \wedge x + t = v) \supset v \in S$$

Whether a given vector is in S is decidable since the set itself is Presburger computable.

6.5.8 End-to-End Example

Let's revisit the tricky case from Section 6.5.2.

```
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>`, we write this as a sequence of vector pairs `t ; k`:

```
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>
```

The corresponding flat language is:

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

The language compiles to the Presburger formula C_4, where

$$\begin{aligned}
C_0 &= \{\texttt{<1,0,0,0,0>}\} \\
C_1 &= \mathsf{post}(C_0, \mathsf{g2}^*) \\
C_2 &= \mathsf{post}(C_1, \mathsf{g1}^*) \\
C_3 &= \mathsf{post}(C_2, \mathsf{g3}^*) \\
C_4 &= \mathsf{post}(C_3, \mathsf{g4}^*)
\end{aligned}$$

If we omit needless existential quantification over rules that only apply a fixed number of times, we can easily generate this set by hand:

$$
\begin{aligned}
C_0 &= \{\texttt{<1,0,0,0,0>}\} \\
C_1 &= \mathsf{post}(C_0, \mathtt{g2}^*) \\
&= \{\texttt{<1,0,0,0,0>}, \\
&\quad \texttt{<0,0,0,1,2>}\} \\
C_2 &= \mathsf{post}(C_1, \mathtt{g1}^*) \\
&= \{\texttt{<1,0,0,0,0>}, \\
&\quad \texttt{<0,0,0,1,2>}, \\
&\quad \texttt{<0,1,1,0,0>}\} \\
C_3 &= \mathsf{post}(C_2, \mathtt{g3}^*) \\
&= \{\texttt{<1,0,0,0,0>}, \\
&\quad \texttt{<0,0,0,1,2>}, \\
&\quad \texttt{<0,1,1,0,0>}\} \cup \\
&\quad \{v \mid \exists n \in \mathbb{N}. v = \texttt{<0,0,1,0,n>}\} \\
C_4 &= \mathsf{post}(C_3, \mathtt{g4}^*) \\
&= \{\texttt{<1,0,0,0,0>}, \\
&\quad \texttt{<0,0,0,1,2>}, \\
&\quad \texttt{<0,1,1,0,0>}\} \cup \\
&\quad \{v \mid \exists n \in \mathbb{N}. v = \texttt{<0,1,1,0,n>}\} \cup \\
&\quad \{v \mid \exists n \in \mathbb{N}. v = \texttt{<0,0,1,0,n>}\}
\end{aligned}
$$

If we further restrict this set to exclude states with nonterminals by intersecting with the set $\{v \mid v_0 = 0 \wedge v_1 = 0\}$, we get the set of well-formed states

$$\{\texttt{<0,0,0,1,2>}\} \cup \{v \mid \exists n \in \mathbb{N}. v = \texttt{<0,0,1,0,n>}\}$$

Or in other words, the contexts $\{b, c, c\}$ and $\{a, c^n\}$ for arbitrary n, as expected.

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 6.5.2, 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. $C' = \mathsf{post}_1(C) =$

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

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

For rule 2,

$$\begin{aligned} C'' &= \mathsf{post}_2(C) \\ &= \{\texttt{<0,0,0,1,2>}\} \cup \\ &\quad \{x \mid \exists n.\texttt{<0,0,0,1,n+2>}\} \cup \\ &\quad \{x \mid \exists n.x = \texttt{<0,0,1,0,n>}\} \cup \\ &\quad \{x \mid \exists n.x\ \texttt{<0,0,1,0,n+1>}\} \end{aligned}$$

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

6.5.9 Adequacy and Correctness

The following lemmas serve as basic sensibility checks on the encoding we have used to devise our decidability proof.

Proposition 6.5.1 (Adequacy)**.** *Modeling generative invariants as vector addition systems is sound and complete with respect to derivability (in the logic) and reachability (in the VAS). That is, formally, if* $|\Sigma_{gen}, \Delta_0| = \langle V, c_0\rangle$ *then* $\Delta_0 \rightarrow *_{\Sigma_{gen}}\theta$*, if and only if* $c_0 \rightsquigarrow^*_V |\theta|$.

This property follows by the standard correspondence between linear logic derivability and vector addition reachability [Kan95].

Proposition 6.5.2 (Soundness)**.** *If the Presburger formula representing a rule* $r : A \multimap B$ *preserving a flat generative invariant* $\langle \Sigma, \Delta_0\rangle$ *has a proof, then for all* Δ, $\Delta_0 \rightsquigarrow^*_\Sigma \Delta, A$ *implies* $\Delta_0 \rightsquigarrow^*_\Sigma \Delta, B$.

Proof Sketch: Let $S = \mathsf{post}^*{}_{|\Sigma|}(|\Delta_0|)$. Let (t, k) be the transition and constraint of $|r|$. By correspondence with vector addition systems, the Presburger formula has a proof if and only if $\mathsf{post}_{|r|}(S) \subseteq S$.

The postset along $|r|$ is $\{v|\exists x \in S.x \geq k \wedge x + t = v\}$.

If we consider the vector respresentation of each side of the rule $|A|$ and $|B|$, we can write t as $|B| - |A|$ and k as $|A|$. So this postset can be written alternatively: $\{v|\exists x \in S.x \geq |A| \wedge x + |B| - |A| = v\}$.

We know $|\Delta, A| \in S$, and we need to show $|\Delta, B| \in S$.

Δ is in the set of vectors $\{v\}$ specified by the postset, because $|\Delta, A|$ is in S, is greater than $|A|$ by translation of contexts into vectors, and $|\Delta, A| + |B| - |A| = |\Delta, B|$. Thus $|\Delta, B| \in S$, as required. □

Proposition 6.5.3 (Completeness)**.** *Converse of soundness. If, for all* Δ, $(\Delta_0 \rightsquigarrow^*_\Sigma \Delta, A)$ *implies* $(\Delta_0 \rightsquigarrow^*_\Sigma \Delta, B)$ *and there is a flat representation of* Σ*, then the Presburger formula representing invariant preservation of* $\langle \Sigma, \Delta_0\rangle$ *along* $r : A \multimap B$ *has a proof.*

Proof Sketch: Let $S = \mathsf{post}^*{}_{|\Sigma|}(|\Delta_0|)$. The Presburger formula has a proof if and only if $\mathsf{post}_r(S) \subseteq S$, i.e. for all v,

$$(\exists x \in S.x \geq |A| \wedge x + |B| - |A| = v) \supset v \in S$$

which is what we need to show.

What we know is that $\forall\Delta.(\Delta_0 \rightarrow^*_\Sigma \Delta, A) \supset (\Delta_0 \rightarrow^*_\Sigma \Delta, B)$, or in VAS terms,

$$|\Delta, A| \in S \supset |\Delta, B| \in S$$

Assume a given vector v, and assume of it the premise of what we need to show: that there exists an x such that $x \in S \wedge x \geq |A| \wedge x + |B| - |A| = v$).

Now instantiate the $|\Delta|$ quantified over in our assumption at $x - |A|$. Now we know $x - |A| + |A| \in S \supset x - |A| + |B| \in S$. We know $x \in S$, so now we know $v \in S$, as required. □

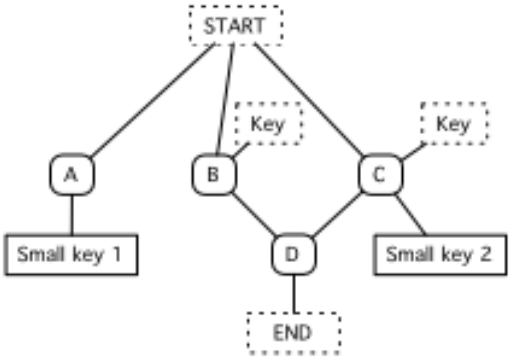
6.6 Conclusion

We have presented three methods for specifying program invariants, and for each discussed their potential for automation and their limitations. We have also presented a decidability proof for checking invariants of programs within a large fragment describing most of the programs we have considered in this thesis.

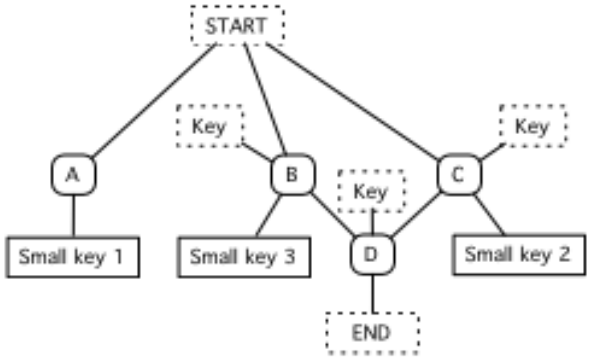
We close this chapter with a discussion of a limitation of all three of these systems, suggesting that there is still a great deal of investigation to do both in terms of automating specification checks and extending the expressiveness of specification languages.

6.6.1 Limitation: Instance-Dependent Invariants

One of our motivating examples for investigating program property proof is allowing the author to state and enforce *playability* constraints. For an example of a playability constraint, consider spatial navigation puzzles, such as the dungeon layouts in which locked doors separate some rooms and keys may be found in some rooms. Gareth Rees has studied this dungeon structure [8] in the context of The Legend of Zelda: Ocarina of Time [Nin98]; we replicate their diagram from Figure 3 here:



Dungeon A: may become unsolvable



Dungeon B: always solvable

In this diagram, rooms (depicted with capidal letters) with "Key" labels require a key (any small key, indistinguishable) to open them, and rooms with "Small key" labels

[8] Article available at `http://garethrees.org/2004/12/01/ocarina-of-time/`.

contain such a key. For the dungeon to be *solvable* means that every room can eventually be reached by a player starting in the "Start" node.

We can easily give a program specifying the space of player actions for such a dungeon, which presumably include moving from room to room, picking up keys, and unlocking rooms that require a key when we have one:

```
door : type.
room : type.

entity : type.
player : entity.
small_key : entity.

adjacent room door room : bwd.
% defined by domain instance

locked door : pred.
open door : pred.

at entity room : pred.
holding entity entity : pred.

move : at player R * adjacent R D R' * open D -o at player R'.
open_door : $at player R * adjacent R D R' * locked D
            * holding player small_key
              -o open D.
pickup_key : $at player R * at small_key R
              -o holding player small_key.
```

Defining "accessibility" between locations requires some kind of recursive specification, which could perhaps be stated as a backward-chaining linear logic program or as a generative invariant of some kind. But note that any invariant that would describe solvability of a dungeon depends crucially on the specific initial configuration, since it is really a property of that configuration as well as of the program—and none of the techniques we have discussed so far encompass such a statement. Temporal logics, such as those used in model checking, present promising techniques that could account for properties like these, wherein it is feasible to exhaustively check the possible configuration space [CGP99]. In lieu of a concrete solution, however, we posit this example as a potential benchmark for game invariant specification techniques going forward.