# Proving Liveness of Parameterized Programs

Azadeh Farzan
University of Toronto

Zachary Kincaid
Princeton University

Andreas Podelski
University of Freiburg

## Abstract

Correctness of multi-threaded programs typically requires that they satisfy liveness properties. For example, a program may require that no thread is starved of a shared resource, or that all threads eventually agree on a single value. This paper presents a method for proving that such liveness properties hold. Two particular challenges addressed in this work are that (1) the correctness argument may rely on global behaviour of the system (e.g., the correctness argument may require that all threads collectively progress towards "the good thing" rather than one thread progressing while the others do not interfere), and (2) such programs are often designed to be executed by *any number* of threads, and the desired liveness properties must hold regardless of the number of threads that are active in the program.

## 1. Introduction

Many multi-threaded programs are designed to be executed in parallel by an arbitrary number of threads. A challenging and practically relevant problem is to verify that such a program is correct no matter how many threads are running.

Let us consider the example of the *ticket mutual exclusion protocol*, pictured in Figure 1. This protocol is an idealized version of the one used to implement spin-locks in the Linux kernel. The protocol maintains two natural-typed variables: `s` (the *service number*) and `t` (the *ticket number*), which are both initially zero. A fixed but unbounded number of threads simultaneously execute the protocol, which operates as follows. First, the thread acquires a ticket by storing the current value of the ticket number into a local variable `m` and incrementing the ticket number (atomically). Second, the thread waits for the service number to reach `m` (its ticket value), and then enters its critical section. Finally, the thread leaves its critical section by incrementing the service number, allowing the thread with the next ticket to enter.

Mutual exclusion, a safety property, is perhaps the first property that comes to mind for this protocol: no two threads should be in their critical sections at the same time. But one of the main reasons that the ticket protocol came to replace simpler implementations of spin-locks in the Linux kernel was because it satisfies *non-starvation* [10] (a liveness property): no thread that acquires a ticket waits forever to enter its critical section (under the fairness assumption that every thread is scheduled to execute infinitely often).



*LICS '16*, July 05 - 08, 2016, New York, NY, USA.

ACM 978-1-4503-4391-6/16/07…$15.00.
http://dx.doi.org/10.1145/2933575.2935310

```
global nat s, t
local nat m
while(true):
  m=t++    // Acquire a ticket
  while(m>s):  // Busy wait
    skip
           // Critical section
  s++          // Exit critical
```

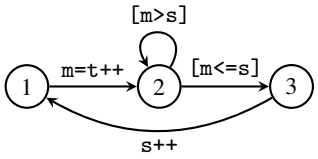


**Figure 1.** Ticket mutual exclusion protocol

Intuitively, the argument for non-starvation in the ticket protocol is obvious: tickets are assigned to threads in sequential order, and whenever a thread exits its critical section, the next thread in the sequence enters. However, it is surprisingly difficult to come up with a formal correctness argument manually, let alone automatically. This paper presents a theoretical foundation for algorithmic verification of liveness properties of multi-threaded programs with any number of threads.

The core of our method is the notion of *well-founded proof spaces*. Well-founded proof spaces are a formalism for proving properties of infinite traces. An *infinite trace* is an infinite sequence of program commands paired with thread identifiers, where a pair $\langle \sigma : \mathtt{i} \rangle$ indicates that the command $\sigma$ is executed by thread `i`. We associate with each well-founded proof space a set of infinite traces that the space proves to be terminating. A well-founded proof space constitutes a proof of program termination if every trace of the program is proved terminating. A well-founded proof space constitutes a proof of a liveness property if every trace of the program that does *not* satisfy the liveness property is proved terminating.

The main technical contribution of the paper is an approach to verifying that a well-founded proof space proves that all program traces terminate. Checking this condition is a language inclusion problem, which is complicated by the fact that the languages consist of words of infinite length, and are defined over an infinite alphabet (since each command must be tagged with an identifier for the thread that executed it). This inclusion problem is addressed in two steps: first, we show how the inclusion between two sets of infinite traces of a particular form can be proven by proving inclusion between two sets of *finite* traces (Theorems 3.8 and 4.2). This is essentially a reduction of infinite trace inclusion to verification of a safety property; the reduction solves the *infinite length* aspect of the inclusion problem. Second, we develop *quantified predicate automata*, a type of automaton suitable for representing these languages that gives a concrete characterization of this safety problem as an emptiness problem (Theorem 4.5). In this context, quantification is used as a mechanism for enforcing behaviour that *all* threads must satisfy. This solves the *infinite alphabet* aspect of the inclusion problem.

The overall contribution of this paper is a formal foundation for automating liveness proofs for parameterized programs. We investigate its theoretical properties and pave the way for future work on exploring efficient algorithms to implement the approach.

### 1.1 Related work

There exist proof systems for verifying liveness properties of parameterized systems (for example, [28]). However, the problem of automatically constructing such proofs has not been explored. To the best of our knowledge, this paper is the first to address the topic of automatic verification of liveness properties of (infinite-state) programs with a parameterized number of threads.

**Parameterized model checking** considers systems that consist of unboundedly many finite-state processes running in parallel [1, 2, 12–14, 24]. In this paper, we develop an approach to the problem of verifying liveness properties of parameterized *programs*, in which processes are infinite state. This demands substantially different techniques than those used in parameterized model checking. The techniques used in this paper are more closely related to *termination analysis* and *parameterized program analysis*.

**Termination analysis** an active field with many effective techniques [7, 9, 11, 19, 23, 31]. One of the goals of the present paper is to adapt the incremental style of termination analysis pioneered by Cook et al. [6, 7] to the setting of parameterized programs. The essence of this idea is to construct a termination argument iteratively via abstraction refinement: First, sample some behaviours of the program and prove that those are terminating. Second, assemble a termination argument for the example behaviours into a candidate termination argument. Third, use a safety checker to prove that the termination argument applies to *all* behaviours of the program. If the safety check succeeds, the program terminates; if not, we can use the counter-example to improve the termination argument.

Termination analyses have been developed for the setting of concurrent programs [8, 22, 26]. Our work differs in two respects. First, our technique handles the case that there are unboundedly many threads operating simultaneously in the system. Second, the aforementioned techniques prove termination using *thread-local* arguments. A thread-local termination argument expresses that each thread individually progresses towards some goal assuming that its environment (formed by the other threads) is either passive or at least does not disrupt its progress. In contrast, the technique proposed in the paper is able to reason about termination that requires coordination between all threads (that is, all threads together progress towards some goal). This enables our approach to prove liveness for programs such as the Ticket protocol (Figure 1): proving that some distinguished thread will eventually enter its critical section requires showing that all *other* threads collectively make progress on increasing the value of the service number until the distinguished thread's ticket is reached.

**Parameterized safety analysis** deals with proving safety properties of infinite state concurrent programs with unboundedly many threads [20, 21, 29, 30]. Safety analysis is relevant to liveness analysis in two respects: (1) In liveness analysis based on abstraction refinement, checking the validity of a correctness argument is reduced to the verification of a safety property [6, 7] (2) An invariant is generally needed in order to establish (or to *support*) a ranking function. Well-founded proof spaces can be seen as an extension of *proof spaces* [15], a proof system for parameterized safety analysis, to prove liveness properties. A more extensive comparison between proof spaces and other methods for parameterized safety analysis can be found in [15].

## 2. Parameterized Program Termination

This section defines parameterized programs and parameterized program termination in a language-theoretic setting.

A *parameterized program* is a multi-threaded program in which each thread runs the same code, and where the number of threads is an input parameter to the system. A parameterized program can be specified by a control flow graph that defines the code that each thread executes. A control flow graph is a directed, labeled graph

$$P = \langle \mathsf{Loc}, \Sigma, \ell_{\mathsf{init}}, \mathsf{src}, \mathsf{tgt} \rangle$$

where $\mathsf{Loc}$ is a set of program locations, $\Sigma$ is a set of program commands, $\ell_{\mathsf{init}}$ is a designated initial location, and $\mathsf{src}, \mathsf{tgt} : \Sigma \to \mathsf{Loc}$ are functions mapping each program command to its source and target location.

Let $P$ be a program as given above. An *indexed command* $\langle \sigma : \mathtt{i} \rangle \in \Sigma \times \mathbb{N}$ of $P$ is a pair consisting of a program command $\sigma$ and an identifier $\mathtt{i}$ for the thread that executes the command.[1] For any natural number $N$, define $\Sigma(N)$ to be the set of indexed commands $\langle \sigma : \mathtt{i} \rangle$ with $\mathtt{i} \in \{1, ..., N\}$.

Let $\Sigma$ be a set of program commands and $N \in \mathbb{N}$ be a natural number. A *trace* over $\Sigma(N)$ is a finite or infinite sequence of indexed commands. We use $\Sigma(N)^*$ to denote the set of all finite traces over $\Sigma(N)$ and $\Sigma(N)^\omega$ to denote the set of infinite traces over $\Sigma(N)$. For a finite trace $\tau$, we use $|\tau|$ to denote the length of $\tau$. For a (finite or infinite) trace $\tau$, we use $\tau_k$ to denote the $k^{th}$ letter of $\tau$ and $\tau[m, n]$ to denote the sub-sequence $\tau_m \tau_{m+1} \cdots \tau_n$. For a finite trace $\tau$, and a (finite or infinite) trace $\tau'$, we use $\tau \cdot \tau'$ to denote the concatenation of $\tau$ and $\tau'$. We use $\tau^\omega$ for the infinite trace obtained by the infinite repeated concatenation of the finite trace $\tau$ $(\tau \cdot \tau \cdot \tau \cdots)$.

For a parameterized program $P$ and a number $N \in \mathbb{N}$, we use $P(N)$ to denote a Büchi automaton that accepts the traces of the $N$-threaded instantiation of the program $P$. Formally, we define $P(N) = \langle Q, \Sigma(N), \Delta, q_0, F \rangle$ where

- $Q = \{1, ..., N\} \to \mathsf{Loc}$ (states are $N$-tuples of locations)
- $\Delta = \{(q, \langle \sigma : \mathtt{i} \rangle, q') : q(\mathtt{i}) = \mathsf{src}(\sigma) \wedge q' = q[\mathtt{i} \mapsto \mathsf{tgt}(\sigma)]\}$
- $q_0 = \lambda i.\ell_{\mathsf{init}}$ (initially, every thread is at $\ell_{\mathsf{init}}$)
- $F = Q$ (every state is accepting)

We use $\mathcal{L}(P(N))$ to denote the language recognized by $P(N)$, and define the set of traces of $P$ to be $\mathcal{L}(P) = \bigcup_{N \in \mathbb{N}} \mathcal{L}(P(N))$. We call the traces in $\mathcal{L}(P)$ *program traces*.

Fix a set of global variables $\mathsf{GV}$ and a set of local variables $\mathsf{LV}$. For any $N \in \mathbb{N}$, we use $\mathsf{LV}(N)$ to denote a set of *indexed local variables* of the form $l(\mathtt{i})$, where $l \in \mathsf{LV}$, and $\mathtt{i} \in \{1, ..., N\}$. $\mathsf{Var}(N)$ denotes the set $\mathsf{GV} \cup \mathsf{LV}(N)$. We do not fix the syntax of program commands. A *program assertion* (program term) is a formula (term) over the vocabulary of some appropriate theory augmented with a symbol for each member of $\mathsf{GV}$ and $\mathsf{LV}(N)$ (for all $N$). For example, the program term $(\mathtt{x}(1) + \mathtt{y}(2) + \mathtt{z})$ refers to the sum of Thread 1's copy of the local variable $\mathtt{x}$, Thread 2's copy of the local variable $\mathtt{y}$, and the global variable $\mathtt{z}$, and can be evaluated in a program state with at least the threads $\{1, 2\}$; the program assertion $(\mathtt{x}(1) > \mathtt{x}(2))$ is satisfied by any state (with at least the threads $\{1, 2\}$) where Thread 1's value for $\mathtt{x}$ is greater than Thread 2's.

We do not explicitly formalize the semantics of parameterized programs, but will rely on an intuitive understanding of some standard concepts. We write $s \models \varphi$ to indicate that the program state $s$ satisfies the program assertion $\varphi$. We write $s \xrightarrow{\langle \sigma : \mathtt{i} \rangle} s'$ to indicate that $s$ may transition to $s'$ when thread $\mathtt{i}$ executes the command $\sigma$. Lastly, we say that a program state $s$ is initial if the program may begin in state $s$.

A trace

$$\langle \sigma_1 : \mathtt{i}_1 \rangle \langle \sigma_2 : \mathtt{i}_2 \rangle \cdots$$

is said to be *feasible* if there exists a corresponding infinite execu-

[1] In the following, we will use typewriter font $\mathtt{i}$ as a meta-variable that ranges over thread identifiers (so $\mathtt{i}$ is just a natural number, but one that is intended to identify a thread).

tion starting from some initial state $s_0$:

$$s_0 \xrightarrow{\langle \sigma_1 : \mathtt{i}_1 \rangle} s_1 \xrightarrow{\langle \sigma_2 : \mathtt{i}_2 \rangle} \cdots .$$

A trace for which there is *no* corresponding infinite execution is said to be *infeasible*.

Finally, we may give our definition of parameterized program termination as follows:

**Definition 2.1** (Parameterized Program Termination)**.** We say that a parameterized program $P$ *terminates* if every program trace of $P$ is *infeasible*. That is, for every $N$, every $\tau \in \mathcal{L}(P(N))$ is infeasible. ⌟

This definition captures the fact that a counter-example to parameterized termination involves only finitely many threads (i.e., a counter example is a trace $\tau \in \mathcal{L}(P(N))$ for some $N$). This is due to the definition of the set of traces of a parameterized program $\mathcal{L}(P)$ (which is a language over an infinite alphabet) as an infinite union of languages $\mathcal{L}(P(N))$, each over a finite alphabet.

The next two sections concentrate on parameterized program termination. We will return to general liveness properties in Section 5.

## 3. Well-founded Proof Spaces

A well-founded proof space is a formalism for proving parameterized termination by proving that its set of program traces are infeasible. This section defines well-founded proof spaces, establishes a sound proof rule for parameterized program termination, and describes how well-founded proof spaces can be used in an incremental algorithm for proving parameterized program termination.

### 3.1 Overview

We motivate the formal definitions that will follow in this section by informally describing the role of well-founded proof spaces in an incremental strategy (*á la* [6, 7]) for proving termination of parameterized programs. The pseudo-code for this (semi-)algorithm is given in Algorithm 1. The algorithm takes as input a parameterized program $P$ and returns "Yes" if $P$ terminates, "No" if $P$ has a trace that can be proved non-terminating, and "Unknown" if the algorithm encounters a trace it cannot prove to be terminating or non-terminating. (There is also a fourth possibility that the algorithm runs forever, repeatedly sampling traces but never finding a termination argument that generalizes to the whole program).

```
   Input : Parameterized program P
1  B ← ∅                                   /* Initialize the basis, B */
   /* Has every program trace been proved infeasible?                  */
2  while L(P) ⊈ ω(⟪B⟫) do
       /* Sample a possibly-feasible trace                             */
3      Pick τ ∈ L(P) \ ω(⟪B⟫)
4      switch FindInfeasibilityProof(τ) do
5          case Infeasibility proof Π
6              Construct B′ from Π so that τ ∈ ω(⟪B′⟫)
7              B ← B + B′
8          case Feasibility proof Π̄
9              return No                   /* P is non-terminating */
10         otherwise
11             return Unknown              /* Inconclusive */
12 return Yes                              /* P is terminating */
```

**Algorithm 1:** Incremental algorithm for parameterized program termination

Algorithm 1 builds a well-founded proof space by repeatedly sampling traces of $P$, finding infeasibility proofs for the samples, and then assembling the proofs into a well-founded proof space. More precisely, the algorithm builds a *basis* $B$ for a proof space, which can be seen as a finite set of axioms that generates a (typically infinite) well-founded proof space $\langle\!\langle B \rangle\!\rangle$. The well-founded proof space $\langle\!\langle B \rangle\!\rangle$ serves as an infeasibility proof for a set of traces, which is denoted $\omega(\langle\!\langle B \rangle\!\rangle)$ (Definition 3.3). The goal of the algorithm is to construct a basis for a well-founded proof space that proves the infeasibility of *every* program trace (at line 2, $\mathcal{L}(P) \subseteq \omega(\langle\!\langle B \rangle\!\rangle)$): if the algorithm succeeds in doing so, then $P$ terminates.

We will illustrate the operation of this algorithm on the simple example pictured in Figure 2. The algorithm begins with an empty basis $B$ (at line 1): the empty basis generates an empty well-founded proof space $\langle\!\langle B \rangle\!\rangle$ that proves infeasibility of an empty set of traces (i.e., $\omega(\langle\!\langle B \rangle\!\rangle) = \emptyset$). Since the inclusion $\mathcal{L}(P) \subseteq \omega(\langle\!\langle B \rangle\!\rangle)$ does not hold (at line 2), we sample (at line 3) a *possibly-feasible* program trace $\tau \in \mathcal{L}(P) \setminus \omega(\langle\!\langle B \rangle\!\rangle)$ (we delay the discussion of how to verify the inclusion $\mathcal{L}(P) \subseteq \omega(\langle\!\langle B \rangle\!\rangle)$ to Section 4). Suppose that our choice for $\tau$ is the trace pictured in Figure 3(a), in which a single thread (Thread 1) executes the loop forever. This trace is *ultimately periodic*: $\tau$ is of the form $\pi \cdot \rho^\omega$, where $\pi$ (the *stem*) and $\rho$ (the *loop*) are finite traces. Under reasonable assumptions (that we formalize in Section 3.3) we ensure that sample traces (counter-examples to the inclusion $\mathcal{L}(P) \subseteq \omega(\langle\!\langle B \rangle\!\rangle)$) are ultimately periodic. The importance of ultimate periodicity is two-fold: first, ultimately periodic traces have a (non-unique) finite representation: a pair of finite words $\langle \pi, \rho \rangle$. Second, ultimately periodic traces correspond to a simple class of sequential programs, allowing Algorithm 1 to leverage the wealth of techniques that have been developed for proving termination [3, 18, 25] and non-termination [17]. The auxiliary procedure FindInfeasibilityProof denotes an (unspecified) algorithm that uses such techniques to prove feasibility or infeasibility of a given trace.

Suppose that calling FindInfeasibilityProof on the sample trace $\tau$ gives the infeasibility proof pictured in Figure 3(b) and (c). The infeasibility proof has two parts. The first part is an *invariance proof*, which is a Hoare proof of an inductive invariant ($\mathtt{d}(1) > 0$) that supports the termination argument. The second part is a *variance proof*, which is a Hoare proof that (assuming the inductive invariant holds at the beginning of the loop) executing the loop causes the state of the program to decrease in some well-founded order. This well-founded order is expressed by the *ranking formula* $\mathit{old}(\mathtt{x}) > \mathtt{x} \wedge \mathit{old}(\mathtt{x}) \geq 0$ (the post-condition of the variance proof). This formula denotes a (well-founded) binary relation between the state of the program and its *old* state (the program state at the beginning of the loop) that holds whenever the value of x decreases and was initially non-negative. Since there is no infinite descending sequence of program states in this well-founded order, the trace $\tau$ (which executes the loop infinitely many times) is infeasible.

We use the termination proof for $\tau$ to construct a basis $B'$ for a well-founded proof space (at line 6). This is done by breaking the termination proof down into simpler components: the Hoare triples that were used in the invariance and variance proofs, and the ranking formula that was used in the variance proof. The basis $B'$ constructed from Figure 3 is pictured in Figure 4. We then add $B'$ to the incrementally constructed basis $B$ (at line 7) and begin the loop again, sampling another possibly-feasible trace $\tau' \in \mathcal{L}(P) \setminus \omega(\langle\!\langle B \rangle\!\rangle)$.

The incremental algorithm makes progress in the sense that it never samples the same trace twice: if $\tau$ is sampled at some loop iteration, then $\tau \in \omega(\langle\!\langle B \rangle\!\rangle)$ for all future iterations. But in fact, $\omega(\langle\!\langle B \rangle\!\rangle)$ contains infinitely many other traces, whose termination proofs can be derived from the same basic building blocks (Hoare triples and ranking formulas) as $\tau$. For example, $\omega(\langle\!\langle B \rangle\!\rangle)$ contains all traces of the form

$$\langle \mathtt{x{=}pos()} : \mathtt{i} \rangle \langle \mathtt{d{=}pos()} : \mathtt{i} \rangle \big( \langle \mathtt{[x{>}0]} : \mathtt{i} \rangle \langle \mathtt{x{=}x{-}d} : \mathtt{i} \rangle \big)^\omega$$

```
global int x
local int d
x = pos()
d = pos()
while (x > 0):
  x = x - d
```

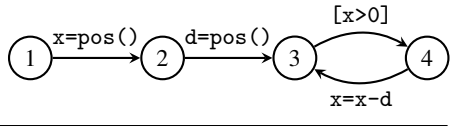


**Figure 2.** Decrement example, pictured along side its control flow graph. The expression `pos()` denotes a non-deterministically generated positive integer, and the command `[x>0]` is an *assumption*; its execution does not change the state of the program, but it can only proceed when `x` is greater than 0.

$$\underbrace{\langle \texttt{x=pos()} : 1\rangle\langle \texttt{d=pos()} : 1\rangle}_{\text{Stem}}\ \underbrace{(\langle \texttt{[x>0]} : 1\rangle\langle \texttt{x=x-d} : 1\rangle)^{\omega}}_{\text{Loop}}$$

(a) An ultimately periodic trace of Figure 2

$\{true\}$
$\langle \texttt{x=pos()} : 1\rangle$
$\{true\}$
$\langle \texttt{d=pos()} : 1\rangle$
$\{\texttt{d}(1) > 0\}$
$\langle \texttt{[x>0]} : 1\rangle$
$\{\texttt{d}(1) > 0\}$
$\langle \texttt{x=x-d} : 1\rangle$
$\{\texttt{d}(1) > 0\}$

(b) Invariance proof

$\{\texttt{d}(1) > 0 \wedge \mathit{old}(\texttt{x}) = \texttt{x}\}$
$\langle \texttt{[x>0]} : 1\rangle$
$\{\texttt{d}(1) > 0 \wedge \mathit{old}(\texttt{x}) = \texttt{x} \wedge \mathit{old}(\texttt{x}) \geq 0\}$
$\langle \texttt{x=x-d} : 1\rangle$
$\{\mathit{old}(\texttt{x}) > \texttt{x} \wedge \mathit{old}(\texttt{x}) \geq 0\}$

(c) Variance proof

**Figure 3.** An ultimately periodic trace and termination proof.

(all of which are, intuitively, infeasible for the same reason as $\tau$). The essential idea is that new Hoare triples and ranking formulas can be deduced from the ones that appear in the basis $B$ by applying some simple inference rules. The resulting collections of Hoare triples and ranking formulas (which are closed under these inference rules) forms a well-founded proof space $\langle\!\langle B \rangle\!\rangle$. Thus in Algorithm 1, well-founded proof spaces serve as a mechanism for *generalizing* infeasibility proofs: they provide an answer to the question *given infeasibility proofs for a finite set of sample traces, how can we re-arrange the ingredients of those proofs to form infeasibility proofs for other traces?*

We will stop our demonstration of Algorithm 1 here, concluding with a listing of the remaining Hoare triples that must be discovered by the algorithm to complete the proof (that is, if those triples are added to the basis $B$, then $\omega(\langle\!\langle B \rangle\!\rangle)$ contains $\mathcal{L}(P)$):

$$\begin{array}{c}
\{\texttt{d}(1) > 0\}\ \langle \texttt{x=pos()} : 2\rangle\ \{\texttt{d}(1) > 0\} \\
\{\texttt{d}(1) > 0\}\ \langle \texttt{d=pos()} : 2\rangle\ \{\texttt{d}(1) > 0\} \\
\{\texttt{d}(1) > 0\}\ \langle \texttt{[x>0]} : 2\rangle\ \{\texttt{d}(1) > 0\} \\
\{\texttt{d}(1) > 0\}\ \langle \texttt{x=x-d} : 2\rangle\ \{\texttt{d}(1) > 0\} \\
\{\mathit{old}(\texttt{x}) \geq \texttt{x}\}\ \langle \texttt{[x>0]} : 1\rangle\ \{\mathit{old}(\texttt{x}) \geq \texttt{x}\} \\
\{\mathit{old}(\texttt{x}) \geq \texttt{x}\}\ \langle \texttt{x=x-d} : 1\rangle\ \{\mathit{old}(\texttt{x}) \geq \texttt{x}\} \\
\{\mathit{old}(\texttt{x}) > \texttt{x}\}\ \langle \texttt{[x>0]} : 1\rangle\ \{\mathit{old}(\texttt{x}) \geq \texttt{x}\} \\
\{\texttt{d}(1) > 0 \wedge \mathit{old}(\texttt{x}) > \texttt{x}\}\ \langle \texttt{x=x-d} : 1\rangle\ \{\mathit{old}(\texttt{x}) > \texttt{x}\}\ .
\end{array}$$

The remainder of this section is organized as follows: in Section 3.2, we give the formal definition of well-founded proof spaces, and describe how a well-founded proof space proves infeasibility of an infinite set of traces. This section treats well-founded proof spaces as a mathematical object, divorcing it from its algorithmic side. In Section 3.3, we describe *regular* well-founded proof spaces, a restricted form of well-founded proof spaces. The key result in this section (Theorem 3.8) is that to prove parameterized program termination, it is sufficient for a *regular* proof space to prove that the ultimately periodic traces of the program terminate.

**Hoare triples**:

$$\begin{array}{c}
\{true\}\ \langle \texttt{x=pos()} : 1\rangle\ \{true\} \\
\{true\}\ \langle \texttt{d=pos()} : 1\rangle\ \{\texttt{d}(1) > 0\} \\
\{\texttt{d}(1) > 0\}\ \langle \texttt{[x>0]} : 1\rangle\ \{\texttt{d}(1) > 0\} \\
\{\texttt{d}(1) > 0\}\ \langle \texttt{x=x-d} : 1\rangle\ \{\texttt{d}(1) > 0\} \\
\{\mathit{old}(\texttt{x}) = \texttt{x}\}\ \langle \texttt{[x>0]} : 1\rangle\ \{\mathit{old}(\texttt{x}) = \texttt{x}\} \\
\{\mathit{old}(\texttt{x}) = \texttt{x}\}\ \langle \texttt{[x>0]} : 1\rangle\ \{\mathit{old}(\texttt{x}) \geq 0\} \\
\{\texttt{d}(1) > 0 \wedge \mathit{old}(\texttt{x}) = \texttt{x}\}\ \langle \texttt{x=x-d} : 1\rangle\ \{\mathit{old}(\texttt{x}) > \texttt{x}\} \\
\{\mathit{old}(\texttt{x}) \geq 0\}\ \langle \texttt{x=x-d} : 1\rangle\ \{\mathit{old}(\texttt{x}) \geq 0\}
\end{array}$$

**Ranking formula**: $\mathit{old}(\texttt{x}) > \texttt{x} \wedge \mathit{old}(\texttt{x}) \geq 0$

**Figure 4.** Basis computed from the termination proof in Figure 3

### 3.2 Formal definition of Well-founded proof spaces

A well-founded proof space is a set of Hoare triples and a set of ranking terms, both closed under certain rules of inference. They serve two roles. First, they are the core of a proof rule for parameterized program termination. A well-founded proof space acts as a termination certificate for a set of infinite traces (Definition 3.3); we may prove that a program $P$ terminates by showing that all traces of $\mathcal{L}(P)$ are contained inside this set. Second, well-founded proof spaces are a mechanism for *proof generalization*: starting from a (finite) *basis* of Hoare triples, we can take the closure of the basis under some simple inference rules to form a well-founded proof space that proves the termination of a larger set of traces (Definition 3.2). We will now define these notions formally.

We begin by formalizing the components of well-founded proof spaces, *Hoare triples* and *ranking formulas*, and their inference rules.

A *Hoare triple*

$$\{\varphi\}\ \langle \sigma : \texttt{i}\rangle\ \{\psi\}$$

consists of an indexed command $\langle \sigma : \texttt{i}\rangle$ and two program assertions $\varphi$ and $\psi$ (the pre- and post-condition of the triple, respectively). We say that such a triple is *valid* if for any pair of program states $s, s'$ such that $s \models \varphi$ and $s \xrightarrow{\langle \sigma : \texttt{i}\rangle} s'$, we have $s' \models \psi$.

We can infer new valid Hoare triples from a set of given ones using the inference rules of proof spaces, namely SEQUENCING, SYMMETRY, and CONJUNCTION [15]. We will recall the definition of these three rules below.

SEQUENCING is a variation of the classical sequencing rule of Hoare logic. For example, we may sequence the two triples

$$\{true\}\ \langle \texttt{d=pos()} : 1\rangle\ \{\texttt{d}(1) > 0\} \text{ and}$$
$$\{\texttt{d}(1) > 0\}\ \langle \texttt{[x>0]} : 1\rangle\ \{\texttt{d}(1) > 0\}$$

to yield

$$\{true\}\ \langle \texttt{d=pos()} : 1\rangle \cdot \langle \texttt{[x>0]} : 1\rangle\ \{\texttt{d}(1) > 0\}\ .$$

Two triples may be sequenced only when the post-condition of the first entails the pre-condition of the first, according to a *combinatorial entailment rule*. The combinatorial entailment relation $\Vdash$ is defined as

$$\varphi_1 \wedge \ldots \wedge \varphi_n \Vdash \psi_1 \wedge \ldots \wedge \psi_m \text{ iff } \{\varphi_1, \ldots, \varphi_n\} \supseteq \{\psi_1, \ldots, \psi_m\}$$

(i.e., $\varphi \Vdash \psi$ iff, viewed as sets of conjuncts, $\varphi$ is a superset of $\psi$). Combinatorial entailment is a weaker version of logical entailment (which is used in the classical sequencing rule in Hoare logic). Our sequencing rule can be written as follows:

SEQUENCING

$$\frac{\{\varphi_0\}\ \tau_0\ \{\varphi_1\} \qquad \varphi_1 \Vdash \varphi_1' \qquad \{\varphi_1'\}\ \tau_1\ \{\varphi_2\}}{\{\varphi_0\}\ \tau_0 \cdot \tau_1\ \{\varphi_2\}}$$

SYMMETRY allows thread identifiers to be substituted uniformly in a Hoare triple. For example, from

$$\{true\}\ \langle \texttt{d=pos()} : 1\rangle\ \{\texttt{d}(1) > 0\}$$

we may derive

$$\{true\}\ \langle \texttt{d=pos()} : 2\rangle\ \{\texttt{d}(2) > 0\}$$

via the symmetry rule. Given a permutation $\pi \in \mathbb{N} \to \mathbb{N}$ and a program assertion $\varphi$, we use $\varphi[\pi]$ to denote the result of substituting each indexed local variable $l(\mathtt{i})$ in $\varphi$ with $l(\pi(\mathtt{i}))$. The symmetry rule may be written as follows:

SYMMETRY

$$\frac{\{\varphi\}\ \ \langle \sigma_1 : \mathtt{i}_1 \rangle \cdots \langle \sigma_n : \mathtt{i}_n \rangle\ \ \{\psi\}}{\{\varphi[\pi]\}\ \ \langle \sigma_1 : \pi(\mathtt{i}_1) \rangle \cdots \langle \sigma_n : \pi(\mathtt{i}_n) \rangle\ \ \{\psi[\pi]\}} \quad \begin{array}{l}\pi : \mathbb{N} \to \mathbb{N} \\ \text{is a permutation}\end{array}$$

CONJUNCTION is precisely the conjunction rule of Hoare logic. For example, from the triples

$$\{\mathtt{d}(1) > 0\}\ \ \langle \mathtt{[x>0]} : 1 \rangle\ \ \{\mathtt{d}(1) > 0\} \text{ and}$$
$$\{\mathit{old}(\mathtt{x}) = \mathtt{x}\}\ \ \langle \mathtt{[x>0]} : 1 \rangle\ \ \{\mathit{old}(\mathtt{x}) \geq 0\}$$

we may derive

$$\{\mathtt{d}(1) > 0 \wedge \mathit{old}(\mathtt{x}) = \mathtt{x}\}\ \ \langle \mathtt{[x>0]} : 1 \rangle\ \ \{\mathtt{d}(1) > 0 \wedge \mathit{old}(\mathtt{x}) \geq 0\}\,.$$

The conjunction rule can be written as follows:

CONJUNCTION

$$\frac{\{\varphi_1\}\ \tau\ \{\psi_1\} \qquad \{\varphi_2\}\ \tau\ \{\psi_2\}}{\{\varphi_1 \wedge \varphi_2\}\ \tau\ \{\psi_1 \wedge \psi_2\}}$$

A *proof space* is defined to be a set of valid Hoare triples that is closed under these three rules [15]. Proof spaces were used in [15] to prove infeasibility of a set of *finite* traces. To form a *well-founded proof space*, which proves infeasibility of a set of *infinite* traces, we enrich a proof space with a set of *ranking formulas*.

A ranking formula is a logical representation of a well-founded order on program states. We suppose that each program variable $\mathtt{x}$ has an associated *old* version $\mathit{old}(\mathtt{x})$ that allows formulas to refer to the value of $\mathtt{x}$ in some "previous" state. Any such formula $\varphi$ can be interpreted as a binary relation $R_\varphi$ on states, with $s\ R_\varphi\ s'$ iff $\varphi$ holds in the interpretation that uses $s$ to interpret the *old* variables and $s'$ to interpret the rest. A *ranking formula* is defined to be a formula $w$ over the program variables and their *old* copies such that the relation $R_w$ is a well-founded order.

The only inference rule that we consider for ranking formulas is a *symmetry* rule: if $w$ is a ranking formula and $\pi : \mathbb{N} \to \mathbb{N}$ is a permutation of thread identifiers, then $w[\pi]$ is a ranking formula.

We may now define well-founded proof spaces formally:

**Definition 3.1** (Well-founded proof space)**.** A *well-founded proof space* $\langle \mathscr{H}, \mathscr{W} \rangle$ is a pair consisting of a set of Hoare triples $\mathscr{H}$ and a set of ranking formulas $\mathscr{W}$ such that $\mathscr{H}$ is closed under SEQUENCING, SYMMETRY, and CONJUNCTION, and $\mathscr{W}$ is closed under permutations of thread identifiers. ⌟

We may present a well-founded proof as the closure of some *basis* (perhaps constructed from termination proofs of some small set of sample traces). Formally,

**Definition 3.2.** Let $H$ be a set of valid Hoare triples, and let $W$ be a set of ranking formulas. $H$ and $W$ *generate* a well-founded proof space $\langle\!\langle H, W \rangle\!\rangle$, defined to be the smallest well-founded proof space $\langle \mathscr{H}, \mathscr{W} \rangle$ such that $H \subseteq \mathscr{H}$ and $W \subseteq \mathscr{W}$. We say that $\langle H, W \rangle$ is a *basis* for $\langle \mathscr{H}, \mathscr{W} \rangle$. ⌟

The fact that $\langle\!\langle H, W \rangle\!\rangle$ is well-defined (i.e., contains only *valid* Hoare triples and ranking functions) follows immediately from the soundness of the inference rules for well-founded proof spaces.

We associate with each well-founded proof space the set of all infinite traces that it proves infeasible. Intuitively, a well-founded proof space proves that a trace $\tau$ is infeasible by exhibiting a ranking formula $w$ and a decomposition of $\tau$ into (infinitely many) finite segments $\tau = \tau_0\tau_1\tau_2\cdots$ such that for each $i$, the program state decreases in the order $w$ along each segment $\tau_i$. More formally,

**Definition 3.3.** Let $\langle \mathscr{H}, \mathscr{W} \rangle$ be a well-founded proof space. We define the set $\omega(\mathscr{H}, \mathscr{W})$ of infinite traces recognized by $\langle \mathscr{H}, \mathscr{W} \rangle$ to be the set of infinite traces $\tau = \langle \sigma_1 : \mathtt{i}_1 \rangle \langle \sigma_2 : \mathtt{i}_2 \rangle \cdots \in \Sigma(N)^\omega$ such that $\{\mathtt{i}_k : k \in \mathbb{N}\}$ is finite and there exists a sequence of naturals $\{\alpha_k\}_{k \in \mathbb{N}}$, and a ranking formula $w \in \mathscr{W}$ such that:

1. For all $k \in \mathbb{N}$, $\alpha_k < \alpha_{k+1}$
2. For all $k \in \mathbb{N}$, there exists some formula $\varphi$ such that
$$\{\mathit{true}\}\ \ \tau[1, \alpha_k]\ \ \{\varphi\} \in \mathscr{H}, \text{ and}$$
$$\{\varphi \wedge \bigwedge_{x \in \mathsf{Var}(N)} \mathit{old}(x) = x\}\ \ \tau[\alpha_k + 1, \alpha_{k+1}]\ \ \{w\} \in \mathscr{H}$$

⌟

The fundamental property of interest concerning the definition of $\omega(\mathscr{H}, \mathscr{W})$ is the following soundness theorem:

**Theorem 3.4** (Soundness)**.** Let $\langle \mathscr{H}, \mathscr{W} \rangle$ be a well-founded proof space. Then every infinite trace in $\omega(\mathscr{H}, \mathscr{W})$ is infeasible. ⌟

Theorem 3.4 is the basis of our proof rule for termination: for a given program $P$, if we can derive a well-founded proof space $\langle \mathscr{H}, \mathscr{W} \rangle$ such that $\omega(\mathscr{H}, \mathscr{W})$ contains all the traces of $P$, then $P$ terminates. This proof rule justifies the soundness of Algorithm 1.

### 3.3 Ultimately periodic traces

Algorithm 1 relies on an auxiliary procedure FindInfeasibilityProof to prove infeasibility of sample traces (counter-examples to the inclusion $\mathcal{L}(P) \subseteq \omega(\langle\!\langle B \rangle\!\rangle)$). An attractive way of implementing FindInfeasibilityProof is to use existing sequential program termination techniques [3, 18, 25] that are very good at proving termination by synthesizing ranking functions for programs of a restricted form, namely so-called *lasso programs*. To take advantage of these techniques, we must ensure that the sample traces given to FindInfeasibilityProof are *ultimately periodic*, so that they may be represented by lasso programs. This section defines a *regularity* condition on well-founded proof spaces that enables us to ensure that ultimately periodic counter-examples always exist.

An ultimately periodic trace is an infinite trace of the form $\pi \cdot \rho^\omega$, where $\pi$ and $\rho$ are finite traces. Such a trace corresponds to a *lasso program*, a sequential program that executes the sequence $\pi$ followed by $\rho$ inside of a **while** loop (since only finitely many threads are involved in $\pi$ and $\rho$, the local variables of each thread may be renamed apart).

The question in this sub-section is *how can we prove parameterized program termination while sampling only the counter-example traces to the sufficiency of the proof argument that are ultimately periodic*? Phrased differently, *is proving termination of ultimately periodic traces enough to prove parameterized program termination?* The (sequential) program to the right illustrates the potential pitfall: even though every ultimately periodic trace of the program is infeasible, the program does not terminate.

```
x=0
while(true):
  i = 0
  while(i < x):
    i++
  x++
```

We place restrictions on well-founded proof spaces so that any (suitably restricted) well-founded proof space that proves termination of all ultimately periodic program traces inevitably proves termination of all program traces (i.e., if $\omega(\mathscr{H}, \mathscr{W})$ includes all ultimately periodic traces in $\mathcal{L}(P)$, it inevitably contains all of $\mathcal{L}(P)$). These restrictions are somewhat technical, and can be skipped on a first reading.

First, we exclude Hoare triples in which local variables "spontaneously appear", such as $\mathtt{x}(2)$ in:

$$\{\mathit{true}\}\ \ \langle \mathtt{x = 0} : 1 \rangle\ \ \{\mathtt{x}(1) = \mathtt{x}(2) \vee \mathtt{x}(1) = 0\}$$

This triple is valid, but the appearance of $\mathtt{x}(2)$ in the post-condition is arbitrary. This technical restriction is formalized by well-formed Hoare triples:

**Definition 3.5** (Well-formed Hoare triple)**.** A Hoare triple

$$\{\varphi\}\ \ \tau\ \ \{\psi\}$$

is *well-formed* if for each $\mathtt{i} \in \mathbb{N}$ such that an indexed local variable of the form $x(\mathtt{i})$ appears in the post-condition $\psi$, then either $\mathtt{i}$ executes some command along $\tau$ or $x(\mathtt{i})$ or some other indexed local variable $y(\mathtt{i})$ with the same index $\mathtt{i}$ appears in the pre-condition $\varphi$. ⌟

The second restriction we make is to require that the well-founded proof space is generated by a finite basis in which there are no "weak" Hoare triples. There are two types of weakness we prohibit. First, we exclude Hoare triples with conjunctive post-conditions

$$\{\varphi\} \ \tau \ \{\psi_1 \wedge \psi_2\}$$

because such a triple can be derived from the pair

$$\{\varphi\} \ \tau \ \{\psi_1\} \text{ and } \{\varphi\} \ \tau \ \{\psi_2\}$$

via the CONJUNCTION rule. Second, we exclude Hoare triples for traces of length greater than one

$$\{\varphi\} \ \tau \cdot \langle \sigma : \mathtt{i} \rangle \ \{\psi\}$$

because such a triple can be derived from the pair

$$\{\varphi\} \ \tau \ \{\varphi'\} \text{ and } \{\varphi'\} \ \langle \sigma : \mathtt{i} \rangle \ \{\psi\}$$

(for some choice of $\varphi'$) via the SEQUENCING rule. We formalize these restrictions with *basic* Hoare triples:

**Definition 3.6** (Basic Hoare triple)**.** A Hoare triple

$$\{\varphi\} \ \langle \sigma : \mathtt{i} \rangle \ \{\psi\}$$

is *basic* if it is valid, well-formed, and the post-condition $\psi$ is atomic in the sense that it cannot be constructed by conjoining two other formulas. ⌟

We call a well-founded proof space that meets all of these technical restrictions *regular*. Formally:

**Definition 3.7** (Regular)**.** We say that a well-founded proof space $\langle \mathscr{H}, \mathscr{W} \rangle$ is *regular* if there exists a *finite* set of *basic* Hoare triples $H$ and a *finite* set of ranking formulas $W$ such that $\langle H, W \rangle$ generates $\langle \mathscr{H}, \mathscr{W} \rangle$. ⌟

The justification for calling such proof spaces regular is that if $\langle \mathscr{H}, \mathscr{W} \rangle$ is regular, then $\omega(\mathscr{H}, \mathscr{W})$ is "nearly" $\omega$-regular, in the sense that $\omega(\mathscr{H}, \mathscr{W}) \cap \Sigma(N)^\omega$ is $\omega$-regular (accepted by a Büchi automaton) for all $N \in \mathbb{N}$.

Finally, we state the main result of this sub-section: regular well-founded proof spaces guarantee the existence of ultimately periodic counter-examples. More precisely, if there is a sample program trace that *cannot* be proved terminating by a given regular well-founded proof space, then there is also an *ultimately periodic* counter-example.

**Theorem 3.8.** Let $P$ be a parameterized program and let $\langle \mathscr{H}, \mathscr{W} \rangle$ be a regular well-founded proof space. If every *ultimately periodic* program trace $\pi \cdot \rho^\omega \in \mathcal{L}(P)$ is included in $\omega(\mathscr{H}, \mathscr{W})$, then every program trace $\tau \in \mathcal{L}(P)$ is included in $\omega(\mathscr{H}, \mathscr{W})$. ⌟

*Proof.* For any set of traces $L \subseteq \bigcup_N \Sigma(N)^\omega$, we define $UP(L)$ to be the set of ultimately periodic traces that belong to $L$:

$$UP(L) = \{\tau \in L : \exists \pi, \rho \in \Sigma(N)^*.\tau = \pi\rho^\omega)\} \ .$$

We must prove that if $UP(\mathcal{L}(P)) \subseteq \omega(\mathscr{H}, \mathscr{W})$ then $\mathcal{L}(P)) \subseteq \omega(\mathscr{H}, \mathscr{W})$. We suppose that $UP(\mathcal{L}(P)) \subseteq \omega(\mathscr{H}, \mathscr{W})$ and (recalling the definition $\mathcal{L}(P) = \bigcup_N \mathcal{L}(P(N))$) prove that for all $N \in \mathbb{N}$, the inclusion $\mathcal{L}(P(N)) \subseteq \omega(\mathscr{H}, \mathscr{W})$ holds.

Let $N$ be an arbitrary natural number. Since $UP(\mathcal{L}(P(N))) \subseteq UP(\mathcal{L}(P)) \subseteq \omega(\mathscr{H}, \mathscr{W})$ we have

$$UP(\mathcal{L}(P(N))) \cap \Sigma(N)^\omega \subseteq \omega(\mathscr{H}, \mathscr{W}) \cap \Sigma(N)^\omega \ .$$

Since (by definition) $\mathcal{L}(P(N)) \subseteq \Sigma(N)^\omega$, we can simplify:

$$UP(\mathcal{L}(P(N))) \subseteq \omega(\mathscr{H}, \mathscr{W}) \cap \Sigma(N)^\omega \ . \tag{1}$$

It is a well known fact that if $L_1$ and $L_2$ are $\omega$-regular languages (over a finite alphabet), then $UP(L_1) \subseteq L_2$ implies $L_1 \subseteq L_2$. The language $\mathcal{L}(P(N))$ is $\omega$-regular by definition, so if we can show that $\omega(\mathscr{H}, \mathscr{W}) \cap \Sigma(N)^\omega$ is $\omega$-regular, then Inclusion (1) implies $\mathcal{L}(P(N)) \subseteq \omega(\mathscr{H}, \mathscr{W}) \cap \Sigma(N)^\omega$ and thus the desired result $\mathcal{L}(P(N)) \subseteq \omega(\mathscr{H}, \mathscr{W})$.

It remains only to show that $\omega(\mathscr{H}, \mathscr{W}) \cap \Sigma(N)^\omega$ is $\omega$-regular. Here we will sketch the intuition *why there exists* a Büchi automaton that recognizes $\omega(\mathscr{H}, \mathscr{W}) \cap \Sigma(N)^\omega$. Since $\langle \mathscr{H}, \mathscr{W} \rangle$ is regular, every Hoare triple in $\mathscr{H}$ is well-formed: this can be proved by induction on the derivation of the triple from the (well-formed) basis. As a result, there are only finitely many program assertions that are relevant to the acceptance condition of a trace $\tau \in \Sigma(N)^\omega$ in $\omega(\mathscr{H}, \mathscr{W})$. Intuitively, we can construct from this finite set of relevant assertions the finite state space of a Büchi automaton that recognizes $\omega(\mathscr{H}, \mathscr{W}) \cap \Sigma(N)^\omega$. □

*Discussion of Theorem 3.8.* The example program above shows that it would not be sound to prove program termination by proving termination of only its ultimately periodic program traces. However, it *is* sound to check sufficiency of a candidate regular well-founded proof space by inspecting only the ultimately periodic program traces. This soundness boils down to the fact that each infinite execution involves only finitely many threads; more technically, the premise of our proof rule (the inclusion between two sets of traces over an infinite alphabet) is equivalent to the validity of an infinite number of inclusions between $\omega$-regular languages over finite alphabets.

## 4. Checking Proof Spaces

The previous section defines a new proof rule for proving termination of parameterized programs: given a parameterized program $P$, if there is some well-founded proof space $\langle \mathscr{H}, \mathscr{W} \rangle$ such that $\omega(\mathscr{H}, \mathscr{W})$ contains every trace of $P$, then $P$ terminates. This section addresses two problems: (1) *how can we verify that the premise of the proof rule holds?*, and (2) *how can we generate an ultimately periodic counter-example if it does not?* The key idea in this section is to reduce the problem of checking the premise (an inclusion problem for sets of infinite traces over an unbounded alphabet) to a non-reachability problem for a particular type of abstract machine (namely, *quantified predicate automata*).

The first step in our reduction to non-reachability is to reduce the inclusion $\mathcal{L}(P) \subseteq \omega(\mathscr{H}, \mathscr{W})$ to an inclusion problem on finite traces. By Theorem 3.8, we know that it is sufficient to check that the ultimately periodic traces of $\mathcal{L}(P)$ are included in $\omega(\mathscr{H}, \mathscr{W})$. Ultimately periodic traces can be represented as finite traces which we call *lassos*. A lasso is a finite trace of the form $\tau\$\rho$, where $\tau, \rho \in \Sigma(N)^*$ (for some $N$) and $\$$ is a special character not appearing in $\Sigma(N)$. A lasso $\tau\$\rho$ can be seen as a finite representation of the ultimately periodic trace $\tau \cdot \rho^\omega$. Note, however, that the correspondence between lassos and ultimately periodic traces is not one-to-one: an ultimately periodic trace $\tau\rho^\omega$ is represented by infinitely many lassos, for example $\tau\$\rho, \tau\$\rho\rho, \tau\rho\$\rho$, and so on.

For a set of traces $L$, we define its lasso language as

$$\$(L) = \{\tau\$\rho : \tau \cdot \rho^\omega \in L\}$$

It is easy to show (using Theorem 3.8) that the inclusion $\mathcal{L}(P) \subseteq \omega(\mathscr{H}, \mathscr{W})$ holds if and only if $\$(\mathcal{L}(P)) \subseteq \$(\omega(\mathscr{H}, \mathscr{W}))$. However, it is *not* easy to give a direct definition of $\$(\omega(\mathscr{H}, \mathscr{W}))$ that lends itself to recognition by an automaton of some variety. Instead, we give an alternate lasso language $\$(\mathscr{H}, \mathscr{W})$ that is not exactly equal to $\$(\omega(\mathscr{H}, \mathscr{W}))$, but (as we will see in the following) is suitable for our purpose:

**Definition 4.1.** Let $\langle \mathscr{H}, \mathscr{W} \rangle$ be a well-founded proof space. Define $\$(\mathscr{H}, \mathscr{W})$ to be the set of lassos $\tau\$\rho$ such that there is some $N \in \mathbb{N}$ so that $\tau, \rho \in \Sigma(N)^*$ and there exists some assertion $\varphi$ and some ranking formula $w \in \mathscr{W}$ such that:

i) $\{\mathit{true}\}\ \ \tau\ \ \{\varphi\} \in \mathscr{H}$

ii) $\{\varphi \wedge \bigwedge_{x \in \mathsf{Var}(N)} \mathit{old}(x) = x\}\ \ \tau\ \ \{w\} \in \mathscr{H}$ ⌟

Note that $\$(\mathscr{H}, \mathscr{W})$ is neither a subset nor a superset of the set of lassos $\$(\omega(\mathscr{H}, \mathscr{W}))$ that correspond to ultimately periodic words in $\omega(\mathscr{H}, \mathscr{W})$. In fact, $\$(\mathscr{H}, \mathscr{W})$ may even contain lassos $\tau\$\rho$ such that $\tau \cdot \rho^\omega$ is feasible: consider for example the lasso $\langle \texttt{y = 1 : 1}\rangle\$\langle \texttt{x = x - y : 1}\rangle\langle \texttt{y = -1 : 1}\rangle$: a well-founded proof space can prove that `x` decreases across the loop of this lasso, but this holds only for the *first iteration* of the loop, and says nothing of subsequent iterations. Despite this, if the inclusion $\$(\mathcal{L}(P)) \subseteq \$(\mathscr{H}, \mathscr{W})$ holds, then every trace of $P$ is proved infeasible by the well-founded proof space $\langle \mathscr{H}, \mathscr{W}\rangle$. The intuition behind this argument is that if the inclusion $\$(\mathcal{L}(P)) \subseteq \$(\mathscr{H}, \mathscr{W})$ holds, then for any ultimately periodic trace $\tau \cdot \rho^\omega$ of $\mathcal{L}(P)$ *every* representation of $\tau \cdot \rho^\omega$ as a lasso is included in $\$(\mathcal{L}(P))$, and thus in $\$(\mathscr{H}, \mathscr{W})$.

**Theorem 4.2** (Inclusion Soundness)**.** Let $P$ be a parameterized program, and let $\langle \mathscr{H}, \mathscr{W}\rangle$ be a regular well-founded proof space. If $\$(\mathcal{L}(P)) \subseteq \$(\mathscr{H}, \mathscr{W})$, then $\mathcal{L}(P) \subseteq \omega(\mathscr{H}, \mathscr{W})$. ⌟

*Proof.* Suppose that the inclusion $\$(\mathcal{L}(P)) \subseteq \$(\mathscr{H}, \mathscr{W})$ holds. By Theorem 3.8, it is sufficient to prove that every ultimately periodic trace of $\mathcal{L}(P)$ is in $\omega(\mathscr{H}, \mathscr{W})$. So let $\tau \cdot \rho^\omega \in \mathcal{L}(P)$, and we will prove that $\tau \cdot \rho^\omega \in \omega(\mathscr{H}, \mathscr{W}))$.

Since $\tau\rho^\omega \in \mathcal{L}(P)$, we must have $\tau\rho^n\$\rho^k \in \$(\mathcal{L}(P)) \subseteq \$(\mathscr{H}, \mathscr{W})$ for all naturals $n$ and positive naturals $k$. From the membership of $\tau\rho^n\$\rho^k$ in $\$(\mathscr{H}, \mathscr{W})$ and the definition of $\$(\mathscr{H}, \mathscr{W})$, there must exist some program assertion $\varphi_{n,k}$ and some ranking formula $w_{n,k} \in \mathscr{W}$ such that:

$$\{\mathit{true}\}\ \ \tau\rho^n\ \ \{\varphi_{n,k}\} \in \mathscr{H}, \text{ and}$$

$$\{\varphi_{n,k} \wedge \bigwedge_{x \in \mathsf{Var}(N)} \mathit{old}(x) = x\}\ \ \rho^k\ \ \{w_{n,k}\} \in \mathscr{H}$$

Define an equivalence relation $\sim$ on the set of pairs $(n, m) \in \mathbb{N}^2$ such that $n < m$ by defining $(n, m) \sim (n', m')$ iff the ranking formulas $w_{n,m-n}$ and $w_{n',m'-n'}$ are equal. Since the set of ranking formulas $\{w_{n,k} \in \mathscr{W} : n, k \in \mathbb{N} \wedge k \geq 1\}$ is finite (following the same reasoning as in the proof of Theorem 3.8), the equivalence relation $\sim$ has finite index. We use $[w]$ to denote the equivalence class consisting of all $(n, m)$ such that $w_{n,m-n} = w$. By Ramsey's theorem [27], there is some ranking formula $w$ and some infinite set of naturals $D \subseteq \mathbb{N}$ such that for all $d, d' \in D$ with $d < d'$, we have $(d, d') \in [w]$.

We conclude that $\tau\rho^\omega \in \omega(\mathscr{H}, \mathscr{W})$ by observing (c.f. Definition 3.3) that there is an infinite sequence of naturals $\{\alpha_i\}_{i \in \mathbb{N}}$ defined by

$$\alpha_i = |\tau| + d_i \cdot |\rho|$$

(where $d_i$ is the $i^{th}$ smallest element of $D$) such that the following hold:

i) For any $i \in \mathbb{N}$, since (by definition) $d_i < d_{i+1}$, we have
$$\alpha_i = |\tau| + d_i \cdot |\rho| < |\tau| + d_{i+1} \cdot |\rho| = \alpha_{i+1}$$

ii) Let $i \in \mathbb{N}$, and define
$$n = (\alpha_i - |\tau|)/|\rho|$$
$$k = (\alpha_{i+1} - \alpha_i)/|\rho| \ .$$
Observe that:
$$\tau\rho^\omega[1, \alpha_i] = \tau \cdot \rho^n$$
$$\tau\rho^\omega[\alpha_i + 1, \alpha_{i+1}] = \rho^k \ .$$
Recalling that $w = w_{n,k}$, it holds that

$$\{\mathit{true}\}\ \ \tau\rho^n\ \ \{\varphi_{n,k}\} \in \mathscr{H}$$
$$\{\varphi_{n,k} \wedge \bigwedge_{x \in \mathsf{Var}(N)} \mathit{old}(x) = x\}\ \ \rho^k\ \ \{w_{n,k}\} \in \mathscr{H} \qquad \square$$

**Remark 4.3.** We note that the reverse of the Inclusion Soundness theorem does not hold: if $\mathcal{L}(P) \subseteq \omega(\mathscr{H}, \mathscr{W})$, it is not necessarily the case that $\$(\mathcal{L}(P)) \subseteq \$(\mathscr{H}, \mathscr{W})$. ⌟

### 4.1 Quantified Predicate Automata

The previous section establishes that a sufficient condition for verifying the premise $\mathcal{L}(P) \subseteq \omega(\mathscr{H}, \mathscr{W})$ of our proof rule (an inclusion problem for sets of infinite traces) is to verify the inclusion $\$(\mathcal{L}(P)) \subseteq \$(\mathscr{H}, \mathscr{W})$ (an inclusion problem for sets of finite traces). In this section, we define *quantified predicate automata*, a class of automata that are capable of recognizing the difference $\$(\mathcal{L}(P)) \setminus \$(\mathscr{H}, \mathscr{W})$. This allows us to characterize the problem of checking the inclusion $\$(\mathcal{L}(P)) \subseteq \$(\mathscr{H}, \mathscr{W})$ as a safety problem: non-reachability of an accepting configuration in a quantified predicate automaton (that is, the *emptiness* problem).

Quantified predicate automata (QPA) are infinite-state and recognize finite traces. QPAs extend predicate automata ([15]) with quantification, enabling them to recognize the lasso language $\$(\mathcal{L}(P))$. Predicate automata are themselves an infinite-state generalization of alternating finite automata [4, 5]. Our presentation of QPA will follow the presentation of predicate automata from [15].

Fix an enumeration $\{i_0, i_1, ...\}$ of variable symbols. Every quantified predicate automaton is equipped with a *finite relational vocabulary* $\langle Q, ar\rangle$, consisting of a finite set of predicate symbols $Q$ and a function $ar : Q \to \mathbb{N}$ that maps each predicate symbol to its arity. We use $\mathcal{F}(Q, ar)$ to denote the set of positive first-order formulas over the vocabulary $\langle Q, ar\rangle$, defined as follows:

$$\varphi, \psi \in \mathcal{F}(Q, ar) ::= \ \ q(i_{j_1}, ..., i_{j_{ar(q)}}) \mid i_j = i_k \mid i_j \neq i_k$$
$$\mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \forall i_j.\varphi \mid \exists i_j.\varphi$$

Quantified predicate automata are defined as follows:

**Definition 4.4** (Quantified predicate automata)**.** A *quantified predicate automaton* (QPA) is a 6-tuple $A = \langle Q, ar, \Sigma, \delta, \varphi_{\mathsf{start}}, F\rangle$ where

- $\langle Q, ar\rangle$ is a finite relational vocabulary,
- $\Sigma$ is a finite alphabet,
- $\varphi_{\mathsf{start}} \in \mathcal{F}(Q, ar)$ is a sentence over the vocabulary $\langle Q, ar\rangle$,
- $F \subseteq Q$ is a set of accepting predicate symbols, and
- $\delta : Q \times \Sigma \to \mathcal{F}(Q, ar)$ is a transition function that satisfies the property that for any $q \in Q$ and $\sigma \in \Sigma$, the free variables of the formula $\delta(q, \sigma)$ belong to the set $\{i_0, ..., i_{ar(q)}\}$. The transition function $\delta$ can be seen as a symbolic rewrite rule
$$q(i_1, ..., i_{ar(q)}) \xrightarrow{\langle \sigma : i_0 \rangle} \delta(q, \sigma) \ ,$$
so the free variable restriction enforces that all variables on the right-hand-side are bound on the left-hand-side. ⌟

A QPA $A = \langle Q, ar, \Sigma, \delta, \varphi_{\mathsf{start}}, F\rangle$ defines an infinite-state non-deterministic transition system, with transitions labeled by indexed commands. The configurations of the transition system are the set of finite structures over the vocabulary $\langle Q, ar\rangle$. That is, a configuration $\mathcal{C}$ of $A$ consists of a finite universe $U^{\mathcal{C}} \subseteq_{\mathsf{fin}} \mathbb{N}$ (where $U^{\mathcal{C}}$ should be interpreted as a set of thread identifiers) along with an interpretation $q^{\mathcal{C}} \subseteq (U^{\mathcal{C}})^{ar(q)}$ for each predicate symbol $q \in Q$. A configuration $\mathcal{C}$ is *initial* $\mathcal{C} \models \varphi_{\mathsf{start}}$, and *accepting* if for all $q \notin F$, $q^{\mathcal{C}} = \emptyset$. Given $A$-configurations $\mathcal{C}$ and $\mathcal{C}'$, $\sigma \in \Sigma$, and $\mathtt{k} \in U^{\mathcal{C}}$, $\mathcal{C}$ transitions to $\mathcal{C}'$ on reading $\langle \sigma : \mathtt{k}\rangle$, written $\mathcal{C} \xrightarrow{\sigma:\mathtt{k}} \mathcal{C}'$, if $\mathcal{C}$ and $\mathcal{C}'$ have the same universe ($U^{\mathcal{C}} = U^{\mathcal{C}'}$), and for all predicate symbols $q \in Q$ and all $\langle \mathtt{i}_1, ..., \mathtt{i}_{ar(q)}\rangle \in q^{\mathcal{C}}$, we have

$$\mathcal{C}' \models \delta(q, \sigma)[i_0 \mapsto \mathtt{k}, i_1 \mapsto \mathtt{i}_1, ..., i_{ar(q)} \mapsto \mathtt{i}_{ar(q)}] \ .$$

For a concrete example of a transition, suppose that

$$\delta(p, a) = p(i_1, i_2) \vee (i_0 \neq i_1 \wedge q(i_2)) \ .$$

To make variable binding more explicit, we will write this rule in

the form

$$\delta(p(i,j), \langle a : k \rangle) = p(i,j) \vee (k \neq i \wedge q(j)) \ .$$

For example, if $\mathcal{C}$ is a configuration with $\mathcal{C} \models p(3,4)$, then a transition $\mathcal{C} \xrightarrow{a:1} \mathcal{C}'$ is possible only when $\mathcal{C}' \models p(3,4) \vee (1 \neq 3 \wedge q(4))$.

QPAs read input traces from right to left. A trace

$$\tau = \langle \sigma_1 : \mathtt{i}_1 \rangle \cdots \langle \sigma_n : \mathtt{i}_n \rangle$$

is accepted by $A$ if there is a sequence of configurations $\mathcal{C}_n, ..., \mathcal{C}_0$ such that $\mathcal{C}_n$ is initial, $\mathcal{C}_0$ is accepting, and for each $r \in \{1, ..., n\}$, we have $\mathcal{C}_r \xrightarrow{\sigma_r : \mathtt{i}_r} \mathcal{C}_{r-1}$. We define $\mathcal{L}(A)$ to be the set all traces that are accepted by $A$.

Recall that the goal stated at the beginning of this section was to develop a class of automaton capable of recognizing the difference $\$(\mathcal{L}(P)) \setminus \$(\mathscr{H}, \mathscr{W})$ (for any given parameterized program $P$ and regular well-founded proof space $\langle \mathscr{H}, \mathscr{W} \rangle$), and thereby arrive at a sufficient automata-theoretic condition for checking the premise of the proof rule established in Section 3. The following theorem states that quantified predicate automata achieve this goal.

**Theorem 4.5.** Let $P$ be a parameterized program and a let $\langle \mathscr{H}, \mathscr{W} \rangle$ be a regular well-founded proof space. Then there is a QPA that accepts $\$(\mathcal{L}(P)) \setminus \$(\mathscr{H}, \mathscr{W})$. ⌟

The proof of this theorem proceeds in three steps: (I) $\$(\mathcal{L}(P))$ is recognizable by a QPA (Proposition 4.6), (II) $\$(\mathscr{H}, \mathscr{W})$ is recognizable by a QPA (Proposition 4.6), and (III) QPAs are closed under Boolean operations (Proposition 4.8). Starting with step (I), we need the following proposition.

**Proposition 4.6.** Let $P$ be a parameterized program. Then there is a QPA $\mathcal{A}(P)$ that accepts $\$(\mathcal{L}(P))$. ⌟

*Proof.* Let $P = \langle \mathsf{Loc}, \Sigma, \ell_{\mathsf{init}}, \mathsf{src}, \mathsf{tgt} \rangle$ be a parameterized program. For a word $\tau \in \Sigma(N)^*$ and a thread $\mathtt{i}$, define $\tau|_{\mathtt{i}}$ to be a the sub-sequence of $\tau$ consisting of the commands executed by thread $\mathtt{i}$. A word $\tau\$\rho$ is a lasso of $P$ if for each thread $\mathtt{i}$, (1) $\tau|_{\mathtt{i}}$ corresponds to a path in $P$, and (2) $\rho|_{\mathtt{i}}$ corresponds to a loop in $P$. We construct the QPA $\mathcal{A}(P) = \langle Q, ar, \Sigma, \delta, \varphi_{\mathsf{start}}, F \rangle$ as follows:

- $Q = \mathsf{Loc} \cup (\mathsf{Loc} \times \mathsf{Loc}) \cup \{\overline{\$}\}$, where $\$$ is a nullary predicate symbol and the rest are monadic. The intuitive interpretation of propositions over this vocabulary are as follows:
  - A trace is accepted starting from a configuration $\mathcal{C}$ with $\mathcal{C} \models \overline{\$}$ if the next letter is not $\$$. This predicate is used to enforce the condition that the loop of a lasso is not empty.
  - For each $\ell \in \mathsf{Loc}$ and each thread $\mathtt{i}$, a trace $\tau$ is accepted starting from a configuration $\mathcal{C}$ with $\mathcal{C} \models \ell(\mathtt{i})$ if $\tau|_{\mathtt{i}}$ corresponds to a path in $P$ ending at $\ell$.
  - For each $\ell_1, \ell_2 \in \mathsf{Loc}$ and each thread $\mathtt{i}$, a trace $\tau\$\rho$ is accepted starting from a configuration $\mathcal{C}$ with $\mathcal{C} \models \langle \ell_1, \ell_2 \rangle(\mathtt{i})$ if $\tau|_{\mathtt{i}}$ corresponds to a path in $P$ ending at $\ell_1$ and $\rho|_{\mathtt{i}}$ corresponds to a path in $P$ from $\ell_1$ to $\ell_2$.
- $\varphi_{\mathsf{start}} = \overline{\$} \wedge \forall i. \bigvee_{\ell \in \mathsf{Loc}} \langle \ell, \ell \rangle(i)$
- $F = \{\ell_{\mathsf{init}}\}$ (the automaton accepts when every thread returns to the initial location)

The transition function $\delta$ is defined as follows.

For any location $\ell_1$, command $\sigma$, and thread $\mathtt{i}$, if $\mathsf{tgt}(\sigma) = \ell_1$ and $\mathtt{i}$ is at $\ell_1$, then reading $\langle \sigma : \mathtt{i} \rangle$ causes thread $\mathtt{i}$ to move from $\ell_1$ to $\mathsf{src}(\sigma)$ while other threads stay put:

$$\delta(\langle \ell_1, \ell_2 \rangle(i), \langle \sigma : j \rangle) = (i = j \wedge \langle \mathsf{tgt}(\sigma), \ell_2 \rangle(i)) \vee (i \neq j \wedge \langle \ell_1, \ell_2 \rangle(i))$$

$$\delta(\ell_1(i), \langle \sigma : j \rangle) = (i = j \wedge \mathsf{tgt}(\sigma)(i)) \vee (i \neq j \wedge \ell_1(i))$$

For any location $\ell_1$, command $\sigma$, and thread $\mathtt{i}$, if thread $\mathtt{i}$ is at $\ell_1$ and $\ell_1 \neq \mathsf{tgt}(\sigma)$, then the automaton rejects when it reads $\langle \sigma : \mathtt{i} \rangle$, but stays put when executing the command of another thread:

$$\delta(\langle \ell_1, \ell_2 \rangle(i), \langle \sigma : j \rangle) = (i \neq j \wedge \langle \ell_1, \ell_2 \rangle(i))$$

$$\delta(\ell_1(i), \langle \sigma : j \rangle) = (i \neq j \wedge \ell_1(i)) \ .$$

Upon reading $\$$, the automaton transitions from $\langle \ell_1, \ell_1 \rangle(i)$ to $\ell_1(i)$:

$$\delta(\langle \ell_1, \ell_1 \rangle(i), \langle \$ : j \rangle) = \ell_1(i) \ ;$$

but for $\ell_1 \neq \ell_2$, the automaton rejects:

$$\delta(\langle \ell_1, \ell_2 \rangle(i), \langle \$ : j \rangle) = \mathit{false} \ .$$

Finally, the nullary predicate $\overline{\$}$ ensures that the next letter in the word is not $\$$:

$$\delta(\overline{\$}, \langle \$ : j \rangle) = \mathit{false}$$

$$\delta(\overline{\$}, \langle \sigma : j \rangle) = \mathit{true} \ . \qquad \square$$

Moving on to step (II):

**Proposition 4.7.** Let $\langle \mathscr{H}, \mathscr{W} \rangle$ be a regular well-founded proof space with basis $\langle H, W \rangle$. Then there is a QPA $\mathcal{A}(H, W)$ that accepts $\$(\mathscr{H}, \mathscr{W})$. ⌟

The construction is similar to the construction of a predicate automaton from a proof space [15]. Intuitively, each Hoare triple in the basis of a regular proof space corresponds to a transition of a QPA. For example, the Hoare triple

$$\{\mathtt{d}(1) > 0 \wedge \mathit{old}(\mathtt{x}) = \mathtt{x}\} \ \ \langle \mathtt{x=x-d} : 1 \rangle \ \ \{\mathit{old}(\mathtt{x}) > \mathtt{x}\}$$

corresponds to the transition

$$\delta([\mathit{old}(\mathtt{x}) > \mathtt{x}], \langle \mathtt{x=x-d} : i \rangle) = [\mathtt{d}(1) > 0](i) \wedge [\mathit{old}(\mathtt{x}) = \mathtt{x}] \ .$$

Details can be found in the extended version of this paper [16].

Finally, we conclude with step (III):

**Proposition 4.8.** QPA languages are closed under Boolean operations (intersection, union, and complement). ⌟

The constructions follow the classical ones for alternating finite automata. Again, details can be found in the extended version [16].

### 4.2 QPA Emptiness

We close this section with a discussion of the emptiness problem for quantified predicate automata. First, we observe that the emptiness problem for QPA is undecidable in the general case, since emptiness is undecidable even for quantifier-free predicate automata [15]. In this respect, our method parallels incremental termination provers for sequential programs: the problem of checking whether a candidate termination argument is sufficient is reduced to a safety problem that is undecidable. Although the emptiness problem is undecidable, safety is a relatively well-studied problem for which there are existing logics and algorithmic techniques. In particular, inductive invariants for QPA can serve as certificates of their emptiness. In the remainder of this section we detail *emptiness certificates*, which formalize this idea.

Intuitively, an *emptiness certificate* for a QPA is a positive formula that is entailed by the initial condition, inductive with respect to the transition relation, and that has no models that are accepting configurations. A problem with this definition is that the transition relation is infinitely-branching (we must verify that the emptiness certificate is inductive with respect to the transition relation labeled with any indexed command, of which there are infinitely many). So first we define a symbolic post-condition operator that gives a finite representation of these infinitely many transitions.

Given a QPA $A = \langle Q, ar, \Sigma, \delta, \varphi_{\mathsf{start}}, F \rangle$, we define a *symbolic post-condition* operator $\hat{\delta} : \mathcal{F}(Q, ar) \times \Sigma \to \mathcal{F}(Q, ar)$ as follows:

$$\hat{\delta}(\varphi, \sigma) = \exists i. \hat{\delta}(\varphi, \langle \sigma : i \rangle),$$

where $i$ is a fresh variable symbol not appearing in $\varphi$ and $\hat{\delta}(\varphi, \langle \sigma : i \rangle)$ is the result of substituting each proposition $q(j_1, ..., j_{ar(q)})$ that appears in $\varphi$ with

$$\delta(q, \sigma)[i_0 \mapsto i, i_1 \mapsto j_1, ..., i_{ar(q)} \mapsto j_{ar(q)}] \ .$$

We may now formally define emptiness certificates:

**Definition 4.9.** Let $A = \langle Q, ar, \Sigma, \delta, \varphi_{\mathsf{start}}, F\rangle$ be a QPA. An *emptiness certificate* for $A$ is a positive first-order formula $\varphi \in \mathcal{F}(Q, ar)$ along with proofs of the following entailments:

1. *Initialization*: $\varphi_{\mathsf{start}} \vdash \varphi$
2. *Consecution*: For all $\sigma \in \Sigma$, $\hat{\delta}(\varphi, \sigma) \vdash \varphi$
3. *Rejection*: $\varphi \vdash \bigvee_{q \in Q \setminus F} \exists i_1, ..., i_{ar(q)}.q(i_1, ..., i_{ar(q)})$. ⌟

The following result establishes that that emptiness certificates are a sound proof system for verifying emptiness of a QPA.

**Theorem 4.10.** Let $A = \langle Q, ar, \Sigma, \delta, \varphi_{\mathsf{start}}, F\rangle$ be a QPA. If there is an emptiness certificate for $A$, then $\mathcal{L}(A)$ is empty. ⌟

## 5. Beyond Termination

In the last two sections presented a technique that uses well-founded proof spaces to prove that parameterized programs terminate. This section extends the technique so that it may be used to prove that parameterized programs satisfy general liveness properties. The class of liveness properties we consider are those that are definable in *(thread) quantified linear temporal logic* (QLTL), which extends linear temporal logic with thread quantifiers to express properties of parameterized systems.

Given a finite alphabet $\Sigma$, a QLTL($\Sigma$) formula is built using the connectives of first-order and linear temporal logic, where quantifiers may **not** appear underneath temporal modalities, and where every proposition is either $i = j$ (for some thread variables $i, j$) or $\langle \sigma : i\rangle$ (for some $\sigma \in \Sigma$ and thread variable $i$). A satisfaction relation $\models$ defines when a trace $\tau$ satisfies a QLTL($\Sigma$) formula, using a map $\mu$ to interpret free variables:

$$\begin{aligned}
\tau, \mu \models i = j &\iff \mu(i) = \mu(j)\\
\tau, \mu \models \langle \sigma : i\rangle &\iff \tau_1 = \langle \sigma : \mu(i)\rangle\\
\tau, \mu \models \varphi \mathbf{U} \psi &\iff \exists k \in \mathbb{N}.(\forall i < k.\tau[i, \omega], \mu \models \varphi)\\
&\qquad\qquad \wedge (\tau[k, \omega], \mu \models \psi)\\
\tau, \mu \models \mathbf{X}\varphi &\iff \tau[2, \omega] \models \varphi\\
\tau, \mu \models \exists i.\varphi &\iff \exists \mathtt{i} \in \{1, ..., N\}.\tau, \mu[i \mapsto \mathtt{i}] \models \varphi\\
\tau, \mu \models \varphi \wedge \psi &\iff \tau, \mu \models \varphi \wedge \tau, \mu \models \psi\\
\tau, \mu \models \neg\varphi &\iff \tau, \mu \not\models \varphi
\end{aligned}$$

The rest of the connectives are defined by the usual equivalences ($\forall i.\varphi \equiv \neg\exists i.\neg\varphi$, $\mathbf{F}\varphi \equiv \mathit{true}\mathbf{U}\varphi$, $\mathbf{G}\varphi \equiv \neg\mathbf{F}\neg\varphi$, $\varphi \vee \psi \equiv \neg(\neg\varphi \wedge \neg\psi)$). For a concrete example, the following formula expresses the liveness property of the ticket protocol (Figure 1), "if every thread executes infinitely often, then no thread is starved":

$$\left(\forall i.\mathbf{GF} \bigvee_{\sigma \in \Sigma} \langle \sigma : i\rangle\right) \Rightarrow \left(\forall i.\mathbf{GF}\langle \texttt{[m<=s]} : i\rangle\right)$$

The theorem enabling well-founded proof spaces to verify QLTL($\Sigma$) properties is the following:

**Theorem 5.1.** Let $\Sigma$ be a finite alphabet, and let $\varphi$ be a QLTL($\Sigma$) sentence. There is a QPA $\mathcal{A}(\varphi)$ that recognizes the language:

$$\$(\mathcal{L}(\varphi)) = \{\tau\$\rho \in \bigcup_N \Sigma(N)^\omega : \tau\rho^\omega \models \varphi\}$$ ⌟

This theorem (proved in [16]) allows us to employ a classical idea for temporal verification [32]: to show that every execution of a program satisfies a QLTL property $\varphi$, we show that every program trace that *violates* $\varphi$ is infeasible. Thus, we have the following proof rule: given a QLTL sentence $\varphi$ and a parameterized program $P$, if there exists regular well-founded proof space $\langle \mathscr{H}, \mathscr{W}\rangle$ with basis $\langle H, W\rangle$ such that the language $\mathcal{L}(\mathcal{A}(P) \wedge \mathcal{A}(\neg\varphi) \wedge \neg\mathcal{A}(H, W))$ is empty, then $P$ satisfies $\varphi$.

**Example 5.2.** To illustrate the idea behind Theorem 5.1, we give a manual construction of a QPA for the (negated) liveness property of the ticket protocol. The negated liveness property can be written as a conjunction of a fairness constraint and a negated liveness constraint:

$$\left(\forall i.\mathbf{GF} \bigvee_{\sigma \in \Sigma} \langle \sigma : i\rangle\right) \wedge \left(\exists i.\mathbf{FG}\neg\langle \texttt{[m<=s]} : i\rangle\right)$$

Given a lasso $\tau\$\rho$, the ultimately periodic word $\tau\rho^\omega$ satisfies the above property iff each thread executes some command along $\rho$ (left conjunct) and there is some thread that does *not* execute `[m<=s]` along $\rho$ (right conjunct). We construct a QPA with two monadic predicates *exec* and $\overline{\mathit{enter}}$ and one nullary predicate $\overline{\$}$ such that

- $\tau\$\rho$ is accepted from a configuration $\mathcal{C}$ with $\mathcal{C} \models \mathit{exec}(\mathtt{i})$ iff $\rho$ contains a command of thread `i`,
- $\tau\$\rho$ is accepted from a configuration $\mathcal{C}$ with $\mathcal{C} \models \overline{\mathit{enter}}(\mathtt{i})$ iff $\rho$ does not contain $\langle$`[m<=s] : i`$\rangle$, and
- $\tau$ is accepted from a configuration $\mathcal{C}$ with $\mathcal{C} \models \overline{\$}$ iff $\tau$ does not contain $\langle$`$ : i`$\rangle$ for any thread `i`.

The initial formula of the QPA is $(\forall i.\mathit{exec}(i)) \wedge (\exists i.\overline{\mathit{enter}}(i))$ and the only accepting predicate symbol is $\overline{\$}$. The transition relation of the QPA is as follows:

$$\begin{aligned}
\delta(\overline{\mathit{enter}}(i), \langle \texttt{[m<=s]} : j\rangle) &= i \neq j \wedge \overline{\mathit{enter}}(i)\\
\delta(\overline{\mathit{enter}}(i), \langle \$ : j\rangle) &= \overline{\$}\\
\delta(\overline{\mathit{enter}}(i), \langle \texttt{m=t++} : j\rangle) &= \overline{\mathit{enter}}(i)\\
\delta(\overline{\mathit{enter}}(i), \langle \texttt{s++} : j\rangle) &= \overline{\mathit{enter}}(i)\\
\delta(\overline{\mathit{enter}}(i), \langle \texttt{[m>s]} : j\rangle) &= \overline{\mathit{enter}}(i)\\
\delta(\mathit{exec}(i), \langle \$ : j\rangle) &= \mathit{false}\\
\delta(\overline{\$}, \langle \$ : j\rangle) &= \mathit{false}
\end{aligned}$$

and for all $\sigma \neq \$$,

$$\begin{aligned}
\delta(\mathit{exec}(i), \langle \sigma : j\rangle) &= i = j \vee \mathit{exec}(i)\\
\delta(\overline{\$}, \langle \sigma : j\rangle) &= \overline{\$}\,.
\end{aligned}$$ ⌟

## 6. Discussion

Although well-founded proof spaces are designed to prove termination of parameterized concurrent programs, a natural question is how they relate to existing methods for proving termination of sequential programs. This section investigates this question. We will compare with the method of **disjunctively well-founded transition invariants**, as exemplified by Terminator [6], and the **language-theoretic** approach, as used by Automizer [19].

Terminator, Automizer, and our approach using well-founded proof spaces employ the same high-level tactic for proving termination. The termination argument is constructed incrementally in a sample-synthesize-check loop: first, sample a lasso of the program, then synthesize a candidate termination argument (using a ranking function for that lasso), then check if the candidate argument applies to the whole program. However, they are based on fundamentally different proof principles.

```
i = pos()
if(0 ≤ i ≤ 1):
  i = 2*i - 1
  // i is either 1 or -1
  while(x>0 ∧ z>0):
    x = x + i
    z = z - i
```

Terminator is based on the principle of disjunctively well-founded transition invariants. Terminator proves termination by showing that the *transitive closure* of a program's transition relation is contained inside the union of a finite number of well-founded relations. As a concrete example, consider the program to the right. Assuming that we restrict ourselves to linear ranking functions, well-founded proof spaces (and Automizer) cannot prove that this program terminates, because there is no linear term that decreases at every loop iteration. Terminator can prove

this program terminates by showing that no matter how many iterations of the loop are executed, `x` decreases *or* `z` decreases.

```
flag = true
while(z > 0):
  if(flag):
    x = z
    z = pos()
  else:
    z = x - 1
    x = pos()
  flag = ¬flag
```

Like well-founded proof spaces, Automizer is based on a language-theoretic view of termination. Automizer proves termination by exhibiting a family of Büchi automata, each of which recognizes a language of traces that terminate "for the same reason" (some given ranking function decreases infinitely often), and such that every trace of the program is recognized by one of the automata. Assuming that we restrict ourselves to linear ranking functions, Terminator cannot prove the program to the right terminates because there is no linear disjunctively well-founded relation that includes the *odd* loop iterations. Automizer (and well-founded proof spaces) can prove the program terminates using the linear ranking function `z`, which decreases infinitely often along any infinite trace (at every even loop iteration).

In the case of *non*-parameterized concurrent programs, well-founded proof spaces are equivalent in power to Automizer. Suppose that $P$ is a program that is intended to be executed by a fixed number of threads $N$ (i.e., we are interested only proving that every trace in $P(N)$ terminates). In this case, the premise of the proof rule ($\mathcal{L}(P) \subseteq \omega(\mathscr{H}, \mathscr{W})$) can be checked effectively using algorithms for Büchi automata, due to the fact that both $\mathcal{L}(P(N))$ and $\omega(\mathscr{H}, \mathscr{W}) \cap \Sigma(N)^\omega$ are $\omega$-regular.

To cope with parameterized programs in which the number of threads is arbitrary, Section 4 describes a *lasso* variation of the proof rule (wherein we check $\$(\mathcal{L}(P)) \subseteq \$(\mathscr{H}, \mathscr{W})$ as a means to prove that $\mathcal{L}(P) \subseteq \omega(\mathscr{H}, \mathscr{W})$). The lasso proof rule is strictly weaker than Automizer's, and the above program cannot be verified for the same reason that Terminator fails: there is no ranking function that decreases after odd iterations of the loop. That is, we cannot construct a well-founded proof space such that $\$(\mathscr{H}, \mathscr{W})$ contains $\tau\$\rho^i$ for odd $i$ (where $\tau$ represents the stem `flag = true` and $\rho$ represents one iteration of the **while** loop). There is an interesting connection between the lasso variant of well-founded proof spaces and disjunctively well-founded transition invariants. Terminator checks that the transitive closure of the transition relation is contained inside a given disjunctively well-founded relation by proving safety of a transformed program. The transformed program executes as the original, but (at some point) non-deterministically saves the program state and jumps to another (disconnected) copy of the program, in which at every loop iteration the program asserts that the "saved" and "current" state are related by the disjunctively well-founded relation. Intuitively, this jump corresponds to exactly the \$ marker in lasso languages: the traces that perform the jump in the transformed program can be put in exact correspondence with the traces of the lasso language $\$(\mathcal{L}(P))$.

Thus, well-founded proof spaces relate to both the Terminator and Automizer proof rules. Section 3 is aligned with the language-theoretic view of program termination used by Automizer. Section 4 mirrors the program transformation employed by Terminator to cope with transitive closure.

## 7. Conclusion

This paper introduces well-founded proof spaces, a formal foundation for automated verification of liveness properties for parameterized programs. Well-founded proof spaces extend the incremental termination proof strategy pioneered in [6, 7] to the case of concurrent programs with unboundedly many threads. This paper investigates a logical foundation of an automated proof strategy. We leave for future work the problem of engineering heuristic techniques to make the framework work in practice.

## References


[1] P. A. Abdulla, Y. Chen, G. Delzanno, F. Haziza, C. Hong, and A. Rezine. Constrained monotonic abstraction: a CEGAR for parameterized verification. In *CONCUR*, pages 86–101, 2010.

[2] P. A. Abdulla, B. Jonsson, M. Nilsson, J. d'Orso, and M. Saksena. Regular model checking for LTL(MSO). *STTT*, 14(2):223–241, 2012.

[3] A. R. Bradley, Z. Manna, and H. B. Sipma. Linear ranking with reachability. In *CAV*, pages 491–504, 2005.

[4] J.A. Brzozowski and E. Leiss. On equations for regular languages, finite automata, and sequential networks. *Theoretical Computer Science*, 10(1):19 – 35, 1980.

[5] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, January 1981.

[6] B. Cook, A. Podelski, and A. Rybalchenko. Abstraction refinement for termination. In *SAS*, pages 87–101, 2005.

[7] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, pages 415–426, 2006.

[8] B. Cook, A. Podelski, and A. Rybalchenko. Proving thread termination. In *PLDI*, pages 320–330, 2007.

[9] B. Cook, A. See, and F. Zuleger. Ramsey vs. lexicographic termination proving. In *TACAS*, pages 47–61, 2013.

[10] J. Corbet. Ticket spinlocks. `https://lwn.net/Articles/267968/`.

[11] P. Cousot and R. Cousot. An abstract interpretation framework for termination. In *POPL*, pages 245–258, 2012.

[12] A. Durand-Gasselin, J. Esparza, P. Ganty, and R. Majumdar. Model checking parameterized asynchronous shared-memory systems. *CoRR*, abs/1505.06588, 2015.

[13] Y. Fang, N. Piterman, A. Pnueli, and L. D. Zuck. Liveness with incomprehensible ranking. In *TACAS*, pages 482–496, 2004.

[14] Y. Fang, N. Piterman, A. Pnueli, and L. D. Zuck. Liveness with invisible ranking. In *VMCAI*, pages 223–238, 2004.

[15] A. Farzan, Z. Kincaid, and A. Podelski. Proof spaces for unbounded parallelism. In *POPL*, pages 407–420, 2015.

[16] A. Farzan, Z. Kincaid, and A. Podelski. Proving liveness of parameterized programs (extended version). `http://arxiv.org/abs/1605.02350`, 2016.

[17] A. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R.-G. Xu. Proving non-termination. In *POPL*, pages 147–158, 2008.

[18] M. Heizmann, J. Hoenicke, J. Leike, and A. Podelski. Linear ranking for linear lasso programs. In *ATVA*, pages 365–380, 2013.

[19] M. Heizmann, J. Hoenicke, and A. Podelski. Termination analysis by learning terminating programs. In *CAV*, pages 797–813, 2014.

[20] J. Jaffar and A. E. Santosa. Recursive abstractions for parameterized systems. In *FM*, pages 72–88. 2009.

[21] A. Kaiser, D. Kroening, and T. Wahl. Lost in abstraction: Monotonicity in multi-threaded programs. In *CONCUR*, pages 141–155, 2014.

[22] J. Ketema and A. F. Donaldson. Automatic termination analysis for GPU kernels. In *Workshop on Termination*, pages 50–55, 2014.

[23] W. Lee, B.-Y. Wang, and K. Yi. Termination analysis with algorithmic learning. In *CAV*, pages 88–104, 2012.

[24] A. Pnueli, S. Ruah, and L. D. Zuck. Automatic deductive verification with invisible invariants. In *TACAS*, pages 82–97, 2001.

[25] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, pages 239–251, 2004.

[26] C. Popeea and A. Rybalchenko. Compositional termination proofs for multi-threaded programs. In *TACAS*, pages 237–251, 2012.

[27] F. P. Ramsey. On a problem of formal logic. In *Proc. London Math. Soc.*, volume 30, pages 264–285, 1930.

[28] A. Sánchez and C. Sánchez. Parametrized verification diagrams. In *TIME*, pages 132–141, 2014.

[29] A. Sanchez, S. Sankaranarayanan, C. Sánchez, and B-Y. E. Chang. Invariant generation for parametrized systems using self-reflection. In *SAS*, pages 146–163. Springer, 2012.

[30] M. Segalov, T. Lev-Ami, R. Manevich, R. Ganesan, and M. Sagiv. Abstract transformers for thread correlation analysis. In *APLAS*, pages 30–46, 2009.

[31] C. Urban. Function: An abstract domain functor for termination - (competition contribution). In *TACAS*, pages 464–466, 2015.

[32] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *LICS*, pages 322–331, 1986.