Inference of ranking functions for proving temporal properties by abstract interpretation

Caterina Urban, Antoine Miné

Inference of Ranking Functions for Proving Temporal Properties by Abstract Interpretation[☆]

Caterina Urban[a,b], Antoine Miné[a,c]

[a] *École Normale Supérieure, Paris, France*
[b] *ETH Zürich, Zurich, Switzerland*
[c] *LIP6 - UPMC, Paris, France*

Abstract

We present new static analysis methods for proving *liveness properties* of programs. In particular, with reference to the hierarchy of temporal properties proposed by Manna and Pnueli, we focus on guarantee (i.e., "something good occurs *at least once*") and recurrence (i.e., "something good occurs *infinitely often*") temporal properties.

We generalize the abstract interpretation framework for termination presented by Cousot and Cousot. Specifically, static analyses of guarantee and recurrence temporal properties are systematically derived by abstraction of the program operational trace semantics.

These methods automatically infer *sufficient preconditions* for the temporal properties by reusing existing numerical abstract domains based on piecewise-defined ranking functions. We augment these abstract domains with new abstract operators, including a *dual widening*.

To illustrate the potential of the proposed methods, we have implemented a research prototype static analyzer, for programs written in a C-like syntax, that yielded interesting preliminary results.

Keywords: static analysis, abstract interpretation, liveness, temporal properties, ranking functions, termination

1. Introduction

Software verification addresses the problem of checking that programs satisfy certain properties. Leslie Lamport, in the late 1970s, suggested a classification of program properties into the classes of *safety* and *liveness* properties [1]. The class of safety properties is informally characterized as the class of properties stating that "something *bad* never happens", that is, a program never reaches an

[☆] *Dedicated to the memory of Radhia Cousot.*

Email addresses:
`caterina.urban@inf.ethz.ch` (Caterina Urban), `antoine.mine@lip6.fr` (Antoine Miné)

unacceptable state. The class of liveness properties is informally characterized as the class of properties stating that "something *good* eventually happens", that is, a program *eventually* reaches a desirable state.

Zohar Manna and Amir Pnueli, in the late 1980s, suggested a more fine grained classification of program properties into a hierarchy [2], which distinguishes four basic classes making different claims about the frequency or occurrence of "something good" mentioned in the informal characterizations proposed by Lamport:

- *safety* properties: "something good *always* happens", i.e., the program never reaches an unacceptable state (e.g., partial correctness, mutual exclusion);
- *guarantee* properties: "something good happens *at least once*", i.e., the program *eventually* reaches a desirable state (e.g., total correctness, termination);
- *recurrence* properties: "something good happens *infinitely often*", i.e., the program reaches a desirable state *infinitely often* (e.g., starvation freedom);
- *persistence* properties: "something good *eventually always* happens", i.e., the program eventually reaches and stays in a desirable state (e.g., stabilization).

This paper concerns the verification of programs by static analysis. We set our work in the framework of Abstract Interpretation [3], a general theory of semantic approximation that provides a basis for various successful industrial-scale tools (e.g., Astrée [4]). Abstract Interpretation has to a large extent been concerned with safety properties and has only recently been extended to program termination [5], which is just a particular guarantee property.

In this paper, we generalize the framework proposed by Patrick Cousot and Radhia Cousot for termination [5] and we propose an abstract interpretation framework for proving *guarantee* and *recurrence* temporal properties of programs. Moreover, we present new static analysis methods for inferring *sufficient preconditions* for these temporal properties. Let us consider the program SIMPLE in Figure 1, where the program variables are interpreted in the set of mathematical integers. The first loop is an infinite loop for the values of the variable x greater than or equal to zero: at each iteration the value of x is increased by one. The second loop is an infinite loop for any value of the variable x: at each iteration, the value of x is increased by one or negated when it becomes greater than ten. Given the guarantee property "$x=3$ at least once", where $x=3$ is the desirable state, our approach is able to automatically infer that the property is true if the initial value of x is smaller than or equal to three. Given the recurrence property "$x=3$ infinitely often", our approach is able to automatically infer that the property is true if the initial value of x is strictly negative (i.e., if the first loop is not entered).

Our approach follows the traditional method for proving liveness properties by means of a well-founded argument (i.e., a function from the states of a program to a well-ordered set whose value decreases during program execution). More precisely, we build a well-founded argument for guarantee and recurrence

```
while [1]( 0≤x ) do [2]x := x+1 od
while [3]( true ) do
  if [4]( x≤10 ) then [5]x := x+1 else [6]x := −x fi
od[7]
```

Figure 1: Program SIMPLE.

properties in an incremental way: we start from the desirable program states, where the function has value zero (and is undefined elsewhere); then, we add states to the domain of the function, retracing the program backwards and counting the maximum number of performed program steps as value of the function. Additionally, for recurrence properties, this process is iteratively repeated in order to construct an argument that is also invariant with respect to program execution steps so that even after reaching a desirable state we know that the execution will reach a desirable state again. We formalize these intuitions into *sound* and *complete* guarantee and recurrence semantics that are systematically derived by abstract interpretation of the program operational trace semantics.

In order to achieve effective static analyses, we further abstract these semantics. Specifically, we leverage existing numerical abstract domains based on piecewise-defined ranking functions [6, 7, 8] by introducing new abstract operators, including a *dual widening*. The piecewise-defined ranking functions are attached to the program control points and represent an *upper bound* on the number of program execution steps before the program reaches a desirable state. They are automatically inferred through backward analysis and yield *sufficient preconditions* for the guarantee and recurrence temporal properties. We prove the soundness of the analysis, meaning that all program executions respecting these preconditions indeed satisfy the temporal properties, while a program execution that does not respect these preconditions might or might not satisfy the temporal properties.

To illustrate the potential of our approach, let us consider again the program SIMPLE in Figure 1. Given the guarantee property "$x=3$ at least once", the piecewise-defined ranking function inferred at program control point 1 is:

$$\lambda x.\ \begin{cases} -3x+10 & x<0 \\ -2x+6 & 0\leq x \wedge x\leq 3 \\ \text{undefined} & \text{otherwise} \end{cases}$$

which bounds the wait (from the program control point 1) for the desirable state $x=3$ by $-3x+10$ program execution steps when $x<0$, and by $-2x+6$ execution steps when $0\leq x\wedge x\leq 3$. The analysis is inconclusive when $3<x$. In this case, when $3<x$, the guarantee property is never satisfied. Thus, the precondition $x\leq 3$ induced by the domain of the ranking function is the weakest precondition for "$x=3$ at least once". Given the recurrence property "$x=3$ infinitely often", the piecewise-defined ranking function at program point 1 bounds the wait for the

next occurrence of the desirable state $x=3$ by $-3x+10$ program execution steps:

$$\lambda x.\ \begin{cases} -3x+10 & x<0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

which induces the precondition $x<0$. Indeed, when $0\leq x \wedge x\leq 3$, the desirable state $x=3$ does not occur infinitely often but only once. Again $x<0$ is the weakest precondition for "$x=3$ at least once". At program point 3 (i.e., at the beginning of the second while loop), for both "$x=3$ infinitely at least" and "$x=3$ infinitely often", we get the following piecewise-defined ranking function:

$$\lambda x.\ \begin{cases} -3x+9 & x\leq 3 \\ -3x+72 & 3<x\leq 10 \\ 3x+12 & 10<x \end{cases}$$

which bounds the wait (from the program point **3**) for the next occurrence of $x=3$ by $-3x+9$ execution steps when $x\leq 3$, by $-3x+72$ execution steps when $3<x\leq 10$, and by $3x+12$ execution steps when $10<x$.

Our Contribution.. In summary, this paper makes the following contributions. First, we present an abstract interpretation framework for proving *guarantee* and *recurrence* program temporal properties. In particular, we generalize the framework proposed by Cousot and Cousot for termination [5]. Moreover, by means of piecewise-defined ranking function abstract domains [6, 7, 8], we design new static analysis methods to effectively infer *sufficient preconditions* for these temporal properties, and provide *upper bounds* in terms of program execution steps on the waiting time before a program reaches a desirable state. Finally, we provide a research prototype static analyzer for programs written in a C-like syntax.

Limitations.. In general, liveness properties are used to specify the behavior of *concurrent* programs and are satisfied only under *fairness* hypotheses. In this paper, we model concurrent programs as non-deterministic sequential programs and we assume that the fair scheduler is explicitly represented within the program (e.g., see [9] and Example 16 in Section 9). We plan, as part of our future work, to extend our framework in order to explicitly express and handle fairness properties.

Outline of the Paper.. Section 2 introduces the preliminary notions used in the paper. In Section 3, we give a brief overview of Cousot and Cousot's abstract interpretation framework for termination. In Section 4 and Section 5, we propose a small idealized programming language used to illustrate our work, and a small specification language used to describe guarantee and recurrence properties. The next two sections are devoted to the main contribution of the paper: we formalize our framework for guarantee and recurrence properties in Section 6 and in Section 7, respectively. In Section 8, we present decidable guarantee and recurrence abstractions based on piecewise-defined ranking functions. We describe our prototype static analyzer in Section 9. Finally, Section 10 discusses related work and Section 11 concludes.

Note.. The results described in this paper have been published in [10] and are presented here with many extensions as well as complete proofs. More specifically, with respect to [10], Section 6 and Section 7 have been extended with the complete denotational definitions for the guarantee and recurrence semantics with respect to the programming language proposed in Section 4. Moreover, Section 2 has been extended in order to provide additional background on the notions that are at the foundation of our work, and many additional examples have been supplied throughout the paper to better illustrate our method.

2. Trace Semantics

In order to be independent from the choice of a particular programming language, following [11, 3], we formalize programs as transition systems:

Definition 1 (Transition System). A *transition system* is a pair $\langle \Sigma, \tau \rangle$ where Σ is a (potentially infinite) set of states and the transition relation $\tau \subseteq \Sigma \times \Sigma$ describes the possible transitions between states.

Note that this model allows representing programs with (possibly unbounded) non-determinism. In some cases, a set $\mathcal{I} \subseteq \Sigma$ is designated as the set of *initial states*. The set of *blocking* or *final states* is $\Omega \stackrel{\text{def}}{=} \{s \in \Sigma \mid \forall s' \in \Sigma \colon \langle s, s' \rangle \notin \tau\}$.

We define the following functions to manipulate sets of program states.

Definition 2. Given a transition system $\langle \Sigma, \tau \rangle$, $\mathrm{pre} \colon \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ maps a set of program states $X \in \mathcal{P}(\Sigma)$ to the set of their predecessors with respect to the program transition relation τ:

$$\mathrm{pre}(X) \stackrel{\text{def}}{=} \{s \in \Sigma \mid \exists s' \in X : \langle s, s' \rangle \in \tau\} \tag{1}$$

Definition 3. Given a transition system $\langle \Sigma, \tau \rangle$, $\widetilde{\mathrm{pre}} \colon \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ maps a set of states $X \in \mathcal{P}(\Sigma)$ to the set of states whose successors with respect to the program transition relation τ are all in the set X:

$$\widetilde{\mathrm{pre}}(X) \stackrel{\text{def}}{=} \{s \in \Sigma \mid \forall s' \in \Sigma : \langle s, s' \rangle \in \tau \Rightarrow s' \in X\} \tag{2}$$

The *semantics* of a program is a mathematical characterization of all possible behaviors of the program when executed for all possible input data. The semantics generated by a transition system is the set of computations described by the transition system. We formally define this notion below.

Given a set $\mathcal{S}$, the set $\mathcal{S}^n \stackrel{\text{def}}{=} \{s_0 \cdots s_{n-1} \mid \forall i < n : s_i \in \mathcal{S}\}$ is the set of all sequences of exactly n elements from $\mathcal{S}$. We write ε to denote the empty sequence, i.e., $\mathcal{S}^0 \triangleq \{\varepsilon\}$. In the following, let $\mathcal{S}^* \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{N}} \mathcal{S}^n$ be the set of all finite sequences, $\mathcal{S}^+ \stackrel{\text{def}}{=} \mathcal{S}^* \setminus \mathcal{S}^0$ be the set of all non-empty finite sequences, $\mathcal{S}^\omega$ be the set of all infinite sequences, $\mathcal{S}^{+\infty} \stackrel{\text{def}}{=} \mathcal{S}^+ \cup \mathcal{S}^\omega$ be the set of all non-empty finite or infinite sequences and $\mathcal{S}^{*\infty} \stackrel{\text{def}}{=} \mathcal{S}^* \cup \mathcal{S}^\omega$ be the set of all finite or infinite sequences of elements from $\mathcal{S}$. In the following, in order to ease the notation, sequences of a

single element $s \in \mathcal{S}$ are often written omitting the curly brackets, e.g., we write s^ω and $s^{+\infty}$ instead of $\{s\}^\omega$ and $\{s\}^{+\infty}$. We write $\sigma\sigma'$ for the concatenation of two sequences $\sigma,\sigma' \in \mathcal{S}^{+\infty}$ (with $\sigma\varepsilon = \varepsilon\sigma = \sigma$, and $\sigma\sigma' = \sigma$ when $\sigma \in \mathcal{S}^\omega$), $\mathcal{T}^+ \stackrel{\text{def}}{=} \mathcal{T} \cap \mathcal{S}^+$ for the selection of the non-empty finite sequences of $\mathcal{T} \subseteq \mathcal{S}^{+\infty}$, $\mathcal{T}^\omega \stackrel{\text{def}}{=} \mathcal{T} \cap \mathcal{S}^\omega$ for the selection of the infinite sequences of $\mathcal{T} \subseteq \mathcal{S}^{+\infty}$ and $\mathcal{T};\mathcal{T}' \stackrel{\text{def}}{=} \{\sigma s \sigma' \mid s \in \mathcal{S} \wedge \sigma s \in \mathcal{T} \wedge s\sigma' \in \mathcal{T}'\}$ for the merging of sets of sequences $\mathcal{T} \subseteq \mathcal{S}^+$ and $\mathcal{T}' \subseteq \mathcal{S}^{+\infty}$, when a finite sequence in $\mathcal{T}$ terminates with the initial state of a sequence in $\mathcal{T}'$.

Given a transition system $\langle \Sigma,\tau \rangle$, a *trace* is a non-empty sequence of states in Σ determined by the transition relation τ, that is, $\langle s,s' \rangle \in \tau$ for each pair of consecutive states $s,s' \in \Sigma$ in the sequence. Note that, the set of final states Ω and the transition relation τ can be understood as a set of traces of length one and a set of traces of length two, respectively. The set of all traces generated by a transition system is called *partial trace semantics*:

Definition 4 (Partial Trace Semantics). The *partial trace semantics* $\dot{\tau}^{+\infty} \in \mathcal{P}(\Sigma^{+\infty})$ generated by a transition system $\langle \Sigma,\tau \rangle$ is defined as follows:

$$\dot{\tau}^{+\infty} \stackrel{\text{def}}{=} \dot{\tau}^+ \cup \tau^\omega$$

where $\dot{\tau}^+ \in \mathcal{P}(\Sigma^+)$ is the set of finite traces:

$$\dot{\tau}^+ \stackrel{\text{def}}{=} \bigcup_{n>0} \{s_0 \cdots s_{n-1} \in \Sigma^n \mid \forall i < n-1 : \langle s_i, s_{i+1} \rangle \in \tau\}$$

and $\tau^\omega \in \mathcal{P}(\Sigma^\omega)$ is the set of infinte traces:

$$\tau^\omega \stackrel{\text{def}}{=} \{s_0 s_1 \cdots \in \Sigma^\omega \mid \forall i \in \mathbb{N} : \langle s_i, s_{i+1} \rangle \in \tau\}$$

Example 1. Let $\Sigma = \{a,b\}$ and $\tau = \{\langle a,a \rangle, \langle a,b \rangle\}$. The partial trace semantics generated by $\langle \Sigma,\tau \rangle$ is the set of traces $a^{+\infty} \cup a^*b$. ■

In practice, given a transition system $\langle \Sigma,\tau \rangle$, and possibly a set of initial states $\mathcal{I} \subseteq \Sigma$, the traces worth of consideration (start by an initial state in $\mathcal{I}$ and) either are infinite or terminate with a final state in Ω. These traces define the *maximal trace semantics* $\tau^{+\infty} \in \mathcal{P}(\Sigma^{+\infty})$ and represent infinite computations or completed finite computations:

Definition 5 (Maximal Trace Semantics). The *maximal trace semantics* $\tau^{+\infty} \in \mathcal{P}(\Sigma^{+\infty})$ generated by a transition system $\langle \Sigma,\tau \rangle$ is defined as:

$$\tau^{+\infty} \stackrel{\text{def}}{=} \tau^+ \cup \tau^\omega$$

where $\tau^+ \in \mathcal{P}(\Sigma^+)$ is the set of finite traces terminating with a final state in Ω:

$$\tau^+ \stackrel{\text{def}}{=} \bigcup_{n>0} \{s_0 \cdots s_{n-1} \in \Sigma^n \mid \forall i < n-1 : \langle s_i, s_{i+1} \rangle \in \tau,\ s_{n-1} \in \Omega\}$$

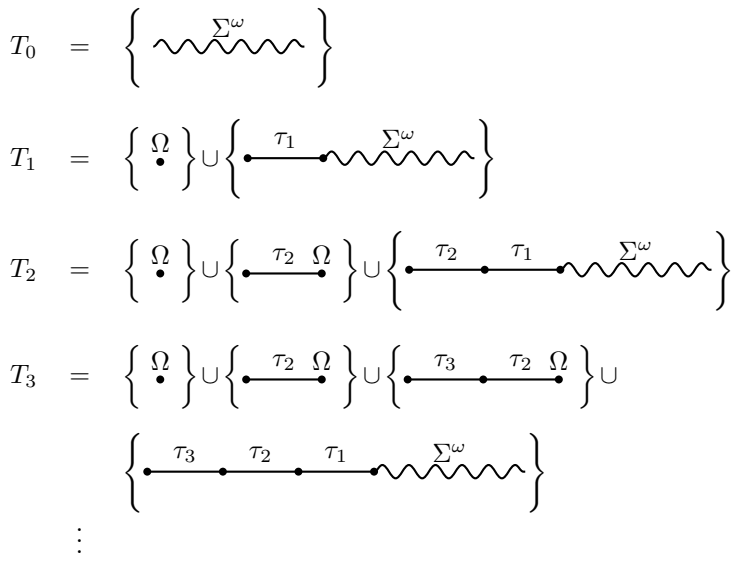


Figure 2: Fixpoint iterates of the maximal trace semantics.

Example 2. The maximal trace semantics generated by the transition system $\langle \Sigma, \tau \rangle$ of Example 1 is the set of traces $a^\omega \cup a^* b$. Note that, unlike the partial trace semantics of Example 1, the maximal trace semantics does not include partial computations, i.e., finite sequences of $a \in \Sigma$. ■

In practice, in case a set of initial states $\mathcal{I} \subseteq \Sigma$ is given, only the traces starting from an initial state $s \in \mathcal{I}$ are considered: $\{s\sigma \in \tau^{+\infty} \mid s \in \mathcal{I}\}$.

In the following, we consider the fixpoint definition of the maximal trace semantics proposed by Patrick Cousot [11]:

$$\begin{aligned} \tau^{+\infty} &= \mathrm{lfp}^{\sqsubseteq}\, \phi^{+\infty} \\ \phi^{+\infty}(T) &\stackrel{\text{def}}{=} \Omega \cup (\tau ; T) \end{aligned} \tag{3}$$

where $\langle \mathcal{P}(\Sigma^{+\infty}), \sqsubseteq, \sqcup, \sqcap, \Sigma^\omega, \Sigma^+ \rangle$ is a complete lattice for the *computational order* is $T_1 \sqsubseteq T_2 \Leftrightarrow T_1^+ \subseteq T_2^+ \wedge T_1^\omega \supseteq T_2^\omega$. In Figure 2, we illustrate the fixpoint iterates. Intuitively, the traces belonging to the maximal trace semantics are built *backwards* by prepending transitions to them: the finite traces are built extending other finite traces from the set of final states Ω, and the infinite traces are obtained by selecting infinite sequences with increasingly longer prefixes forming traces. In particular, the i-th iterate builds all finite traces of length less than or equal to i, and selects all infinite sequences whose prefixes of length i form traces. At the limit we obtain all infinite traces and all finite traces that terminate in Ω.

The maximal trace semantics carries all information about a program. It is the most precise semantics and it fully describes the behavior of a program.

However, to reason about a particular program property, it is not necessary to consider all aspects and details of the program behavior. In fact, reasoning is facilitated by the design of a semantics that abstracts away from irrelevant details about program executions. In particular, rather than deriving program semantics by intuition and justifying them a posteriori, Abstract Interpretation [3] offers an elegant and constructive way to systematically derive *different* program semantics by successive abstractions of the *same* maximal trace semantics.

We illustrate such idea in the following. We first systematically derive a well-adapted semantics for program termination. Then, we derive new program semantics dedicated to guarantee and recurrence properties.

3. Termination Semantics

The traditional method for proving program termination dates back to Alan Turing [12] and Robert W. Floyd [13]. It consists in inferring *ranking functions*, namely functions from program states to elements of a well-ordered set whose value decreases during program execution.

Definition 6 (Ranking Function). Given a transition system $\langle \Sigma, \tau \rangle$, a *ranking function* is a partial function $f : \Sigma \rightharpoonup \mathcal{W}$ from the set of states Σ into a well-ordered set $\langle \mathcal{W}, \leq \rangle$ whose value decreases through transitions between states, that is $\forall s, s' \in \mathrm{dom}(f) : \langle s, s' \rangle \in \tau \Rightarrow f(s') < f(s)$.

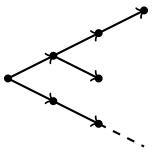
The best known well-ordered sets are the natural numbers $\langle \mathbb{N}, \leq \rangle$ and the ordinals $\langle \mathbb{O}, \leq \rangle$, and the most obvious ranking function maps each program state to the number of program execution steps until termination, or some well-chosen upper bound on this numbers.

In [5], Patrick Cousot and Radhia Cousot prove the existence of a *most precise ranking function* $\tau_{\mathrm{t}} \in \Sigma \rightharpoonup \mathbb{O}$ that can be derived by abstract interpretation of the program maximal trace semantics and can be expressed as a least fixpoint as:

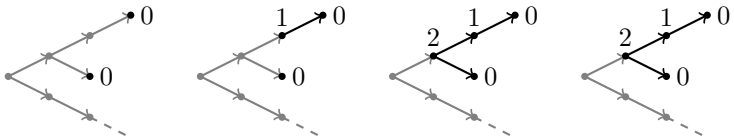
$$\begin{aligned} \tau_{\mathrm{t}} &= \mathrm{lfp}^{\sqsubseteq}_{\dot{\emptyset}}\ \phi_{\mathrm{t}} \\ \phi_{\mathrm{t}}(f) &\stackrel{\text{def}}{=} \lambda s. \begin{cases} 0 & s \in \Omega \\ \sup\{ f(s') + 1 \mid \langle s, s' \rangle \in \tau \} & s \in \widetilde{\mathrm{pre}}(\mathrm{dom}(f)) \\ \text{undefined} & \text{otherwise} \end{cases} \end{aligned} \tag{4}$$

where $\langle \Sigma \rightharpoonup \mathbb{O}, \sqsubseteq \rangle$ forms a partially ordered set for the *computational order* $f_1 \sqsubseteq f_2 \Leftrightarrow \mathrm{dom}(f_1) \subseteq \mathrm{dom}(f_2) \wedge \forall x \in \mathrm{dom}(f_1) : f_1(x) \leq f_2(x)$ and $\mathrm{lfp}^{\sqsubseteq}_{\dot{\emptyset}}$ denotes the least fixpoint greater than or equal to the totally undefined (ranking) function $\dot{\emptyset}$. The most precise ranking function τ_{t} is defined starting from the final states in Ω, where the function has value zero, and retracing the program *backwards* while mapping each program state in Σ definitely leading to a final state (i.e., a program state such that all program traces to which it belong are terminating) to an ordinal in $\mathbb{O}$ representing an upper bound on the number of program execution steps remaining to termination.

Example 3. Let us consider the following trace semantics:



The fixpoint iterates of the most precise ranking function τ_{t} are:



where unlabelled states are outside the domain of the function. ■

The domain of τ_{t} is the set of states from which all program executions terminate; all traces branching from a state $s \in \mathrm{dom}(\tau_{\mathrm{t}})$ terminate in at most $\tau_{\mathrm{t}}(s)$ execution steps, while at least one trace branching from a state $s \notin \mathrm{dom}(\tau_{\mathrm{t}})$ does not terminate:

Theorem 1. *A program terminates for all traces starting from a given set of initial states $\mathcal{I}$ if and only if $\mathcal{I} \subseteq \mathrm{dom}(\tau_t)$.*

Proof. See [5]. □

Intuitively, a ranking function f_1 is more precise than another ranking function f_2 when it is defined over a larger set of program states, that is, it can prove termination for more program states, and when its value is always smaller, that is, the maximum number of program execution steps required for termination is smaller. Thus, we define the *approximation order* between ranking functions as $f_1 \preccurlyeq f_2 \;\Leftrightarrow\; \mathrm{dom}(f_1) \supseteq \mathrm{dom}(f_2) \wedge \forall x \in \mathrm{dom}(f_2) : f_1(x) \leq f_2(x)$. Observe that, the computational order used to define fixpoints and the approximation order often coincide but, in the general case, they are distinct and totally unrelated [14]. We will need to maintain this distinction throughout the rest of this paper.

In [5], Patrick Cousot and Radhia Cousot derive $\tau_{\mathrm{t}} \in \Sigma \rightharpoonup \mathbb{O}$ (cf. Equation 4) by means of successive abstractions of the maximal trace semantics $\tau^{+\infty} \in \mathcal{P}(\Sigma^{+\infty})$ (cf. Equation 3). In the following, we briefly retrace their steps.

We define the *neighborhood* of a sequence $\sigma \in \Sigma^{+\infty}$ in a set of sequences $T \subseteq \Sigma^{+\infty}$ as the set of sequences $\sigma' \in T$ with a common prefix with σ:

$$\mathrm{nbhd}(\sigma, T) \stackrel{\text{def}}{=} \{\sigma' \in T \mid \mathrm{pf}(\sigma) \cap \mathrm{pf}(\sigma') \neq \emptyset\} \tag{5}$$

where $\mathrm{pf} \in \Sigma^{+\infty} \to \mathcal{P}(\Sigma^{+\infty})$ yields the set of prefixes of a sequence $\sigma \in \Sigma^{+\infty}$:

$$\mathrm{pf}(\sigma) \stackrel{\text{def}}{=} \{\sigma' \in \Sigma^{+\infty} \mid \exists \sigma'' \in \Sigma^{*\infty} : \sigma = \sigma'\sigma''\}. \tag{6}$$

A program trace is terminating if and only if it is finite and its neighborhood in the program semantics consists only of finite traces, i.e., the trace terminates independently from the non-deterministic choices made during execution. The corresponding *termination abstraction* $\alpha^{\mathrm{t}}\colon \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma^{+})$ is defined as follows:

$$\alpha^{\mathrm{t}}(T) \stackrel{\text{def}}{=} \{\sigma \in T^{+} \mid \mathrm{nhbd}(\sigma, T^{\omega}) = \emptyset\}. \tag{7}$$

Example 4. Let $T = \{ab, aba, ba, bb, ba^{\omega}\}$ be a set of sequences. Then, its termination abstraction is $\alpha^{\mathrm{t}}(T) = \{ab, aba\}$ since $\mathrm{nbhd}(ab, T^{\omega}) = \emptyset$ and $\mathrm{nbhd}(aba, T^{\omega}) = \emptyset$. In fact, $\mathrm{nbhd}(ab, T^{\omega}) = \mathrm{nbhd}(ab, \{ba^{\omega}\}) = \emptyset$ (i.e., $\mathrm{pf}(ab) \cap \mathrm{pf}(ba^{\omega}) = \emptyset$, cf. Equation 5) and $\mathrm{nbhd}(aba, T^{\omega}) = \mathrm{nbhd}(aba, \{ba^{\omega}\}) = \emptyset$ (i.e., $\mathrm{pf}(aba) \cap \mathrm{pf}(ba^{\omega}) = \emptyset$), while $\mathrm{nbhd}(ba, T^{\omega}) = \mathrm{nbhd}(ab, \{ba^{\omega}\}) = \{ba\} \neq \emptyset$ (i.e., $\mathrm{pf}(ba) \cap \mathrm{pf}(ba^{\omega}) = \{ba\} \neq \emptyset$) and $\mathrm{nbhd}(bb, T^{\omega}) = \mathrm{nbhd}(bb, \{ba^{\omega}\}) = \{b\} \neq \emptyset$ (i.e., $\mathrm{pf}(bb) \cap \mathrm{pf}(ba^{\omega}) = \{b\} \neq \emptyset$). ■

The *termination semantics* $\tau_{\mathrm{t}} \in \Sigma \rightharpoonup \mathbb{O}$ can now be explicitly defined as abstract interpretation of the maximal trace semantics $\tau^{+\infty} \in \mathcal{P}(\Sigma^{+\infty})$:

$$\tau_{\mathrm{t}} \stackrel{\text{def}}{=} \alpha^{\mathrm{rk}}(\alpha^{\mathrm{t}}(\tau^{+\infty})) \tag{8}$$

where the *ranking abstraction* $\alpha^{\mathrm{rk}}\colon \mathcal{P}(\Sigma^{+}) \to (\Sigma \rightharpoonup \mathbb{O})$ is:

$$\alpha^{\mathrm{rk}}(T) \stackrel{\text{def}}{=} \alpha^{\mathrm{v}}(\vec{\alpha}(T)) \tag{9}$$

where the function $\vec{\alpha}\colon \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma \times \Sigma)$ extracts from a set of sequences $T \subseteq \Sigma^{+\infty}$ the smallest transition relation $r \subseteq \Sigma \times \Sigma$ that generates T:

$$\vec{\alpha}(T) \stackrel{\text{def}}{=} \{\langle s, s' \rangle \mid \exists \sigma \in \Sigma^{*}, \sigma' \in \Sigma^{*\infty}\colon \sigma s s' \sigma' \in T\}$$

and where the function $\alpha^{\mathrm{v}}\colon \mathcal{P}(\Sigma \times \Sigma) \to (\Sigma \rightharpoonup \mathbb{O})$ provides the rank of the elements in the domain of a relation $r \subseteq \Sigma \times \Sigma$:

$$\alpha^{\mathrm{v}}(r)s \stackrel{\text{def}}{=} \begin{cases} 0 & \forall s' \in \Sigma\colon \langle s, s' \rangle \notin r \\ \sup\left\{\alpha^{\mathrm{v}}(r)s' + 1 \;\middle|\; s' \in \mathrm{dom}(\alpha^{\mathrm{v}}(r)) \wedge \langle s, s' \rangle \in r\right\} & \text{otherwise} \end{cases}$$

In Section 6.1 and Section 7.1, we will follow the same abstract interpretation approach in order to systematically derive sound and complete semantics for proving guarantee and recurrence temporal properties of programs.

4. A Small Imperative Language

The formal treatment given in the previous chapter is language independent. In the following, for simplicity we consider a while language with some non-deterministic assignments and tests . The variables are statically allocated and the only data type is the set $\mathbb{Z}$ of mathematical integers. Note that our implementation, described in Section 9, actually supports a subset of the C language sufficient to handle real examples from actual benchmarks (e.g., the benchmarks of the International Competition on Software Verification[1]).

[1] http://sv-comp.sosy-lab.org

$$
\begin{array}{lcll}
\textit{aexp} & ::= & X & X \in \mathcal{X} \\
& | & [i_1,i_2] & i_1 \in \mathbb{Z} \cup \{-\infty\},\ i_2 \in \mathbb{Z} \cup \{+\infty\},\ i_1 \leq i_2 \\
& | & -\ \textit{aexp} & \\
& | & \textit{aexp} \diamond \textit{aexp} & \diamond \in \{+,-,*,/\} \\
\\
\textit{bexp} & ::= & \texttt{?} & \\
& | & \texttt{not}\ \textit{bexp} & \\
& | & \textit{bexp}\ \texttt{and}\ \textit{bexp} & \\
& | & \textit{bexp}\ \texttt{or}\ \textit{bexp} & \\
& | & \textit{aexp} \bowtie \textit{aexpr} & \bowtie\ \in \{<,\leq,=,\neq\} \\
\\
\textit{stmt} & ::= & {}^{l}\texttt{skip} & \\
& | & {}^{l}X := \textit{aexp} & l \in \mathcal{L},\ X \in \mathcal{X} \\
& | & \texttt{if}\ {}^{l}\textit{bexp}\ \texttt{then}\ \textit{stmt}\ \texttt{else}\ \textit{stmt}\ \texttt{fi} & l \in \mathcal{L} \\
& | & \texttt{while}\ {}^{l}\textit{bexp}\ \texttt{do}\ \textit{stmt}\ \texttt{od} & l \in \mathcal{L} \\
& | & \textit{stmt}\ \textit{stmt} & \\
\\
\textit{prog} & ::= & \textit{stmt}\ {}^{l} & l \in \mathcal{L}
\end{array}
$$

Figure 3: Syntax of our programming language.

4.1. Language Syntax

In Figure 3, we define inductively the syntax of our programming language.

A program *prog* consists of an instruction followed by a unique label $l \in \mathcal{L}$. Another unique label appears within each instruction. An instruction *stmt* is either a `skip` instruction, a variable assignment, a conditional `if` statement, a `while` loop or a sequential composition of instructions.

Arithmetic expressions *aexp* involve variables $X \in \mathcal{X}$, numeric intervals $[a,b]$ and the arithmetic operators $+$, $-$, $*$, $/$ for addition, subtraction, multiplication, and division. Numeric intervals have constant and possibly infinite bounds, and denote a random choice of a number in the interval. This provides a notion of non-determinism useful to model user input or to approximate arithmetic expressions that cannot be represented exactly in the language. Numeric constants are a particular case of numeric interval. We often write the constant c for the interval $[c,c]$.

Boolean expressions *bexp* are built by comparing arithmetic expressions, and are combined using the boolean `not`, `and`, and `or` operators. The boolean expression `?` represents a non-deterministic choice and is useful to provide a sequential encoding of concurrent programs by modeling a (possibly, but not necessarily, fair) scheduler. Whenever clear from the context, we frequently abuse notation and use the symbol `?` to also denote the numeric interval $[-\infty,+\infty]$.

4.2. Language Semantics

In the following, we instantiate the definition of transition system (cf. Definition 1) with respect to programs written in our small imperative language.

Expression Semantics. An *environment* $\rho \colon \mathcal{X} \to \mathbb{Z}$ maps each program variable $X \in \mathcal{X}$ to its value $\rho(X) \in \mathbb{Z}$. Let $\mathcal{E}$ denote the set of all environments.

The semantics of an arithmetic expression *aexp* is a function $[\![aexp]\!] \colon \mathcal{E} \to \mathcal{P}(\mathbb{Z})$ mapping an environment $\rho \in \mathcal{E}$ to the possible values for the expression *aexp* in the environment. Such semantics is standard, for the sake of completeness its formal definition is given in Appendix A. Note that the set of values for an expression may contain several elements because of the non-determinism in the expressions. It might also be empty due to undefined results (e.g., in case of divisions by zero).

Similarly, the semantics $[\![bexp]\!] \colon \mathcal{E} \to \mathcal{P}(\{\text{true,false}\})$ of boolean expressions *bexp* maps an environment $\rho \in \mathcal{E}$ to the set of all possible truth values for the expression *bexp* in the environment. Such semantics is also standard, and its formal definition is given in Appendix A. In the following, we write `true` and `false` to represent a boolean expression that is always true and always false, respectively.

Transition Systems. A program *state* $s \in \mathcal{L} \times \mathcal{E}$ is a pair consisting of a label $l \in \mathcal{L}$ and an environment $\rho \in \mathcal{E}$, where the ρ defines the values of the program variables at the program point designated by l. Let Σ denote the set of all program states.

The *initial* control point $i[\![stmt]\!] \in \mathcal{L}$ (resp. $i[\![prog]\!] \in \mathcal{L}$) of an instruction *stmt* (resp. a program *prog*) defines where the execution of the instruction (resp. program) starts, and the *final* control point $f[\![stmt]\!] \in \mathcal{L}$ (resp. $f[\![prog]\!] \in \mathcal{L}$) defines where the execution of the instruction *stmt* (resp. program *prog*) ends. The formal definitions are given in Appendix A. A program execution starts at its initial program control point with any possible value for the program variables.

The set of initial states of a program *prog* is $\mathcal{I} \stackrel{\text{def}}{=} \{\langle i[\![prog]\!], \rho\rangle \mid \rho \in \mathcal{E}\}$. The set of final states is $\mathcal{Q} \stackrel{\text{def}}{=} \{\langle f[\![prog]\!], \rho\rangle \mid \rho \in \mathcal{E}\}$.

Remark 1. In Section 2 we defined the final states to have no successors with respect to the transition relation, meaning that the program halts: $\Omega \stackrel{\text{def}}{=} \{s \in \Sigma \mid \forall s' \in \Sigma \colon \langle s, s'\rangle \notin \tau\}$. This is the case when the program successfully terminates by reaching its final label, or when a *run-time error* occurs. For the sake of simplicity, the definition of program final states given in this section ignores possible run-time errors silently halting the program.

Example 5. Let us consider again the program SIMPLE from Figure 1. The set of program environments $\mathcal{E}$ contains functions $\rho \colon \{x\} \to \mathbb{Z}$ mapping the program variable x to any possible value $\rho(x) \in \mathbb{Z}$. The set of program states $\Sigma \stackrel{\text{def}}{=} \{\mathbf{1},\mathbf{2},\mathbf{3},\mathbf{4},\mathbf{5},\mathbf{6},\mathbf{7}\} \times \mathcal{E}$ consists of all pairs of numerical labels and environments; the initial states are $\mathcal{I} \stackrel{\text{def}}{=} \{\langle \mathbf{1}, \rho\rangle \mid \rho \in \mathcal{E}\}$ and the final states are $\mathcal{Q} \stackrel{\text{def}}{=} \{\langle \mathbf{7}, \rho\rangle \mid \rho \in \mathcal{E}\}$.■

We now define the transition relation $\tau \in \Sigma \times \Sigma$. In particular, in Figure 4, we define the transition semantics $\tau[\![stmt]\!] \in \Sigma \times \Sigma$ of each program instruction *stmt*. Given an environment $\rho \in \mathcal{E}$, a program variable $X \in \mathcal{X}$ and a value $v \in \mathbb{Z}$, we denote by $\rho[X \leftarrow v]$ the environment obtained by writing v into X in ρ:

$$\rho[X \leftarrow v](x) = \begin{cases} v & x = X \\ \rho(x) & x \neq X \end{cases}$$

$$
\begin{array}{l}
\tau[\![{}^{l}\texttt{skip}]\!] \stackrel{\text{def}}{=} \{\langle l,\rho\rangle \to \langle f[\![{}^{l}\texttt{skip}]\!],\rho\rangle \mid \rho\in\mathcal{E}\} \\
\tau[\![{}^{l}X := aexp]\!] \stackrel{\text{def}}{=} \{\langle l,\rho\rangle \to \langle f[\![{}^{l}X := aexp]\!],\rho[X\leftarrow v]\rangle \mid \rho\in\mathcal{E},\, v\in[\![aexp]\!]\rho\} \\
\tau[\![\texttt{if } {}^{l}bexp \texttt{ then } stmt_1 \texttt{ else } stmt_2 \texttt{ fi}]\!] \stackrel{\text{def}}{=} \\
\quad \{\langle l,\rho\rangle \to \langle i[\![stmt_1]\!],\rho\rangle \mid \rho\in\mathcal{E},\, \text{true}\in[\![bexp]\!]\rho\} \cup \tau[\![stmt_1]\!] \cup \\
\quad \{\langle l,\rho\rangle \to \langle i[\![stmt_2]\!],\rho\rangle \mid \rho\in\mathcal{E},\, \text{false}\in[\![bexp]\!]\rho\} \cup \tau[\![stmt_2]\!] \\
\tau[\![\texttt{while } {}^{l}bexp \texttt{ do } stmt \texttt{ od}]\!] \stackrel{\text{def}}{=} \\
\quad \{\langle l,\rho\rangle \to \langle i[\![stmt]\!],\rho\rangle \mid \rho\in\mathcal{E},\, \text{true}\in[\![bexp]\!]\rho\} \cup \tau[\![stmt]\!] \cup \\
\quad \{\langle l,\rho\rangle \to \langle f[\![\texttt{while } {}^{l}bexp \texttt{ do } stmt \texttt{ od}]\!],\rho\rangle \mid \rho\in\mathcal{E},\, \text{false}\in[\![bexp]\!]\rho\} \\
\tau[\![stmt_1\ stmt_2]\!] \stackrel{\text{def}}{=} \tau[\![stmt_1]\!] \cup \tau[\![stmt_2]\!]
\end{array}
$$

Figure 4: Transition semantics of instructions *stmt*.

The semantics of a `skip` instruction simply moves control from the initial label of the instruction to its final label. The execution of a variable assignment ${}^{l}X := aexp$ moves control from the initial label of the instruction to its final label, and modifies the current environment in order to assign any of the possible values of *aexp* to the variable X. The semantics of a conditional statement `if` ${}^{l}bexp$ `then` $stmt_1$ `else` $stmt_2$ `fi` moves control from the initial label of the instruction to the initial label of $stmt_1$, if true is a possible value for *bexp*, and to the initial label of $stmt_2$, if false is a possible value for *bexp*; then, $stmt_1$ and $stmt_2$ are executed. Similarly, the execution of a while statement `while` ${}^{l}bexp$ `do` *stmt* `od` moves control from the initial label of the instruction to its final label, if false is a possible value for *bexp*, and to the initial label of $stmt_1$, if true is a possible value for *bexp*; then *stmt* is executed. Note that, control moves from the end of *stmt* to the initial label l of the `while` loop. Finally, the semantics of the sequential combination of instructions $stmt_1$ $stmt_2$ executes $stmt_1$ and $stmt_2$.

The transition relation $\tau\in\Sigma\times\Sigma$ of a program *prog* is defined by the semantics $\tau[\![prog]\!]\in\Sigma\times\Sigma$ of the program as $\tau[\![prog]\!] = \tau[\![stmt\ {}^{l}]\!] \stackrel{\text{def}}{=} \tau[\![stmt]\!]$.

Example 6. Let us consider again the program SIMPLE from Figure 1. The program transition relation $\tau\in\Sigma\times\Sigma$ is defined as follows:

$$
\begin{array}{rl}
\tau \stackrel{\text{def}}{=} & \{\langle \mathbf{1},\rho\rangle \to \langle \mathbf{2},\rho\rangle \mid \rho\in\mathcal{E}\wedge\text{true}\in[\![0\le x]\!]\rho\} \\
& \cup\{\langle \mathbf{2},\rho\rangle \to \langle \mathbf{1},\rho[x\leftarrow\rho(x)-1]\rangle \mid \rho\in\mathcal{E}\} \\
& \cup\{\langle \mathbf{1},\rho\rangle \to \langle \mathbf{3},\rho\rangle \mid \rho\in\mathcal{E}\wedge\text{false}\in[\![0\le x]\!]\rho\} \\
& \cup\{\langle \mathbf{3},\rho\rangle \to \langle \mathbf{4},\rho\rangle \mid \rho\in\mathcal{E}\} \\
& \cup\{\langle \mathbf{4},\rho\rangle \to \langle \mathbf{5},\rho\rangle \mid \rho\in\mathcal{E}\wedge\text{true}\in[\![x\le 10]\!]\rho\} \\
& \cup\{\langle \mathbf{5},\rho\rangle \to \langle \mathbf{3},\rho[x\leftarrow\rho(x)+1]\rangle \mid \rho\in\mathcal{E}\} \\
& \cup\{\langle \mathbf{4},\rho\rangle \to \langle \mathbf{6},\rho\rangle \mid \rho\in\mathcal{E}\wedge\text{false}\in[\![x\le 10]\!]\rho\} \\
& \cup\{\langle \mathbf{6},\rho\rangle \to \langle \mathbf{3},\rho[x\leftarrow-\rho(x)]\rangle \mid \rho\in\mathcal{E}\}
\end{array}
$$

4.3. Denotational Termination Semantics

In the following, we provide a structural definition of the fixpoint termination semantics $\tau_{\text{t}}\in\Sigma\rightharpoonup\mathbb{O}$ (cf. Equation 4) by induction on the syntax of programs

written in our small imperative language.

We partition τ_t with respect to the program control points: $\tau_t \in \mathcal{L} \to (\mathcal{E} \rightharpoonup \mathbb{O})$. In this way, to each program control point $l \in \mathcal{L}$ corresponds a partial function $f : \mathcal{E} \rightharpoonup \mathbb{O}$, and to each program instruction *stmt* corresponds a termination semantics transformer $\tau_t[\![\, stmt]\!] : (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$. Analogously to Equation 4, the ranking function is built *backwards*: each transformer $\tau_t[\![\, stmt]\!] : (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$ takes as input a ranking function whose domain represents the terminating environments at the final control point of *stmt*, and determines a ranking function whose domain represents the terminating environments at the initial control point of *stmt*, and whose value represents an upper bound on the number of program execution steps remaining to termination.

Skip Instruction. The termination semantics of a `skip` instruction takes as input a ranking function $f : \mathcal{E} \rightharpoonup \mathbb{O}$ whose domain represents the terminating environments at the final label of the instruction, and increases its value by one to take into account that from the environments at the initial label of the instruction another program execution step is necessary before termination:

$$\tau_t[\![{}^l\texttt{skip}]\!]f \stackrel{\text{def}}{=} \lambda\rho \in \mathrm{dom}(f).\ f(\rho)+1 \tag{10}$$

Assignment Instruction. Similarly, the termination semantics of a variable assignment ${}^l X := aexp$ takes as input a ranking function $f : \mathcal{E} \rightharpoonup \mathbb{O}$ whose domain represent the terminating environments at the final label of the instruction. The resulting ranking function is defined over the environments that when subject to the variable assignment always belong to the domain of the input ranking function. The value of the input ranking function for these environments is increased by one, to take into account another execution step before termination, and the value of the resulting ranking function is the least upper bound of these values, in order to take non-determinism into account:

$$\tau_t[\![{}^l X := aexp]\!]f \stackrel{\text{def}}{=} \lambda\rho.\ \begin{cases} \sup\{f(\rho[X \leftarrow v])+1 \mid v \in [\![aexp]\!]\rho\} & \\ \qquad [\![aexp]\!]\rho \neq \emptyset \wedge \forall v' \in [\![aexp]\!]\rho :\ \rho[X \leftarrow v'] \in \mathrm{dom}(f) & \\ \text{undefined} & \text{otherwise} \end{cases} \tag{11}$$

Note that all environments yielding a run-time error due to a division by zero do not belong to the domain of the termination semantics of the assignment.

Example 7. Let us consider again the program SIMPLE from Figure 1. We assume that the following ranking function $f : \mathcal{E} \rightharpoonup \mathbb{O}$ is valid at program point **3** during some iterate of the termination semantics:

$$f \stackrel{\text{def}}{=} \lambda\rho.\ \begin{cases} 0 & \rho(x) = 3 \\ \text{undefined} & \text{otherwise} \end{cases}$$

and we consider the assignment $x := x+1$ at program point **5**. The termination semantics of the assignment, given the ranking function, is:

$$\tau_t[\![x := x+1]\!]f \stackrel{\text{def}}{=} \lambda\rho.\ \begin{cases} 1 & \rho(x) = 2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

In particular, note that the function is only defined when $\rho(x)=2$. In fact, when for example $\rho(x)=1$, we have $[\![x+1]\!]\rho=\{2\}$ and $\rho[x\leftarrow 2]\notin \mathrm{dom}(f)$. Similarly, when for example $\rho(x)=3$, we have $[\![x+1]\!]\rho=\{4\}$ and $\rho[x\leftarrow 4]\notin \mathrm{dom}(f)$.

Conditional Instruction. Given a conditional `if` ^{l}*bexp* `then` *stmt*$_1$ `else` *stmt*$_2$ `fi`, its termination semantics takes as input a ranking function $f:\mathcal{E}\rightharpoonup\mathbb{O}$, whose value represents an upper bound on the number of execution steps to termination from the final control point of the instruction. Then, it derives the termination semantics $\tau_{\mathrm{t}}[\![stmt_1]\!]f$ of *stmt*$_1$, in the following denoted by S_1, and the termination semantics $\tau_{\mathrm{t}}[\![stmt_2]\!]f$ of *stmt*$_2$, in the following denoted by S_2. The value of S_1 (respectively, S_2) represents an upper bound on the number of execution steps from the initial control of *stmt*$_1$ (respectively, *stmt*$_2$). The termination semantics of the conditional instruction is defined by means of the ranking function $F_1:\mathcal{E}\rightharpoonup\mathbb{O}$ whose domain is the set of environments $\rho\in\mathcal{E}$ that belong to the domain of S_1 and that must satisfy *bexp*:

$$F_1 \stackrel{\text{def}}{=} \lambda\rho\in\mathrm{dom}(S_1).\begin{cases} S_1(\rho)+1 & [\![bexp]\!]\rho=\{\text{true}\}\\ \text{undefined} & \text{otherwise}\end{cases}$$

and the ranking function $F_2:\mathcal{E}\rightharpoonup\mathbb{O}$ whose domain is the set of environments that belong to the domain of S_2 and that cannot satisfy *bexp*:

$$F_2 \stackrel{\text{def}}{=} \lambda\rho\in\mathrm{dom}(S_2).\begin{cases} S_2(\rho)+1 & [\![bexp]\!]\rho=\{\text{false}\}\\ \text{undefined} & \text{otherwise}\end{cases}$$

and the ranking function $F:\mathcal{E}\rightharpoonup\mathbb{O}$ whose domain is the set of environments that belong to the domain of S_1 and to the domain of S_2, and that due to non-determinism may both satisfy and not satisfy the boolean expression *bexp*:

$$F \stackrel{\text{def}}{=} \lambda\rho\in\mathrm{dom}(S_1)\cap\mathrm{dom}(S_2).\begin{cases} \sup\{S_1(\rho)+1,S_2(\rho)+1\} & [\![bexp]\!]\rho=\{\text{true,false}\}\\ \text{undefined} & \text{otherwise}\end{cases}$$

The value of F_1, F_2, and F represents an upper bound on the execution steps to termination from the initial control point of the conditional instruction when only the first branch is taken, when only the second branch is takes, or when (due to non-determinism) both branches are taken, respectively. The resulting ranking function is defined joining F_1, F_2, and F:

$$\tau_{\mathrm{t}}[\![\texttt{if}\ ^{l}bexp\ \texttt{then}\ stmt_1\ \texttt{else}\ stmt_2\ \texttt{fi}]\!]f \stackrel{\text{def}}{=} F_1\dot{\cup}F_2\dot{\cup}F \tag{12}$$

where $\dot{\cup}$ joins partial functions with disjoint domains: given $f_1:\mathcal{A}\rightharpoonup\mathcal{B}$ and $f_2:\mathcal{A}\rightharpoonup\mathcal{B}$ such that $\mathrm{dom}(f_1)\cap\mathrm{dom}(f_2)=\emptyset$, $(f_1\,\dot{\cup}\,f_2)(x)=f_1(x)$, when $x\in\mathrm{dom}(f_1)$, and $(f_1\dot{\cup}f_2)(x)=f_2(x)$, when $x\in\mathrm{dom}(f_2)$.

Example 8. Let us consider again the program SIMPLE from Figure 1. We consider the conditional statement `if` *bexp* `then` *stmt*$_1$ `else` *stmt*$_2$ `fi` at program

point 4. We assume, given a ranking function $f : \mathcal{E} \rightharpoonup \mathbb{O}$ valid at program point 3 during some iterate, that the termination semantics of $stmt_1$ is defined as:

$$\tau_t[\![stmt_1]\!]f \stackrel{\text{def}}{=} \lambda\rho. \begin{cases} 1 & \rho(x) \leq 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

and that the termination semantics of $stmt_2$ is defined as

$$\tau_t[\![stmt_2]\!]f \stackrel{\text{def}}{=} \lambda\rho. \begin{cases} 3 & 0 \leq \rho(x) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Then, since the boolean expression $bexp$ is $x \leq 10$, the termination semantics of the conditional statement is:

$$\tau_t[\![\texttt{if}\ ^l bexp\ \texttt{then}\ stmt_1\ \texttt{else}\ stmt_2\ \texttt{fi}]\!]f \stackrel{\text{def}}{=} \lambda\rho. \begin{cases} 2 & \rho(x) \leq 0 \\ 4 & 3 < \rho(x) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Instead, if $bexp$ is for example the non-deterministic choice `?`, we have:

$$\tau_t[\![\texttt{if}\ ^l bexp\ \texttt{then}\ stmt_1\ \texttt{else}\ stmt_2\ \texttt{fi}]\!]f \stackrel{\text{def}}{=} \lambda\rho. \begin{cases} 4 & \rho(x) = 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Loop Instruction. The termination semantics of a loop `while` $^l bexp$ `do` $stmt$ `od` takes as input a ranking function $f : \mathcal{E} \rightharpoonup \mathbb{O}$ the domain of which represents the terminating environments at the final label of the instruction (i.e., after exiting the loop), and outputs the ranking function which is defined as a least fixpoint of the function $\phi_t : (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$ within $\langle \mathcal{E} \rightharpoonup \mathbb{O}, \sqsubseteq \rangle$, analogously to Equation 4:

$$\tau_t[\![\texttt{while}\ ^l bexp\ \texttt{do}\ stmt\ \texttt{od}]\!]f \stackrel{\text{def}}{=} \text{lfp}^{\sqsubseteq}_{\dot{\emptyset}}\ \phi_t \tag{13}$$

The function $\phi_t : (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$ takes as input a ranking function $x : \mathcal{E} \rightharpoonup \mathbb{O}$ and adds to its domain the environments for which one more loop iteration is needed before termination. In the following, the termination semantics $\tau_t[\![stmt]\!]x$ of the loop body is denoted by S. The function ϕ_t is defined by means of the ranking function $F_1 : \mathcal{E} \rightharpoonup \mathbb{O}$ whose domain is the set of environments $\rho \in \mathcal{E}$ that belong to the domain of S and that must satisfy $bexp$:

$$F_1 \stackrel{\text{def}}{=} \lambda\rho \in \text{dom}(S). \begin{cases} S(\rho)+1 & [\![bexp]\!]\rho = \{\text{true}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

and the ranking function $F_2 : \mathcal{E} \rightharpoonup \mathbb{O}$ whose domain is the set of environments that belong to the domain of the input function f and that cannot satisfy $bexp$:

$$F_2 \stackrel{\text{def}}{=} \lambda\rho \in \text{dom}(f). \begin{cases} f(\rho)+1 & [\![bexp]\!]\rho = \{\text{false}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

and the ranking function $F\colon \mathcal{E} \rightharpoonup \mathbb{O}$ whose domain is the set of environments that belong to the domain of S and to the domain of the input function f, and that may both satisfy and not satisfy the boolean expression *bexp*:

$$F \stackrel{\text{def}}{=} \lambda\rho{\in}\mathrm{dom}(S){\cap}\mathrm{dom}(f).\ \begin{cases} \sup\{S(\rho){+}1, f(\rho){+}1\} & [\![bexp]\!]\rho = \{\text{true},\text{false}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

The resulting ranking function is defined joining F_1, F_2, and F:

$$\phi_{\mathrm{t}}(x) \stackrel{\text{def}}{=} F_1 \dot{\cup} F_2 \dot{\cup} F \tag{14}$$

Composition Instruction. Finally, the termination semantics of the sequential combination of instructions $stmt_1\ \ stmt_2$, takes as input a ranking function $f\colon \mathcal{E} \rightharpoonup \mathbb{O}$ at the final control point of $stmt_2$, determines from f the termination semantics $\tau_{\mathrm{t}}[\![stmt_2]\!]f$ of $stmt_2$, and feeds it as input to the termination semantics of $stmt_1$ in order to get a ranking function at the initial control point of $stmt_1$:

$$\tau_{\mathrm{t}}[\![stmt_1\ \ stmt_2]\!]f \stackrel{\text{def}}{=} \tau_{\mathrm{t}}[\![stmt_1]\!](\tau_{\mathrm{t}}[\![stmt_2]\!]f) \tag{15}$$

Program Termination Semantics. The termination semantics $\tau_{\mathrm{t}}[\![prog]\!] \in \mathcal{E} \rightharpoonup \mathbb{O}$ of a program *prog* is a ranking function whose domain represents the terminating environments, which is determined taking as input the zero function:

$$\tau_{\mathrm{t}}[\![prog]\!] \ = \ \tau_{\mathrm{t}}[\![stmt\ ^{l}]\!] \stackrel{\text{def}}{=} \tau_{\mathrm{t}}[\![stmt]\!](\lambda\rho.\ 0). \tag{16}$$

Note that, as pointed out in Remark 1, possible run-time errors silently halting the program are ignored. More specifically, all environments leading to run-time errors are discarded and do not belong to the domain of the termination semantics.

In Section 6.2 and Section 7.2, we provide a similar denotation for the guarantee properties semantics defined in Section 6.1 and for the recurrence properties semantics defined in Section 7.1.

5. Program Properties

In general, we define a program *property* as a set of sequences of program states. A program has a certain property if all its traces belong to the property. In this paper, with respect to the hierarchy of program properties proposed in [2], we focus on guarantee ("something good happens *at least once*") and recurrence ("something good happens *infinitely often*") properties. In particular, we consider guarantee and recurrence properties that are expressible by temporal logic.

We assume an underlying specification language, which is used to describe properties of program states. For instance, for our small imperative language, we define inductively the syntax of the state properties as follows:

$$\varphi \ ::= \ bexp \mid l\colon bexp \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \qquad\qquad l \in \mathcal{L}$$

The predicate $l\colon bexp$ allows specifying a program state property at a particular program control point $l \in \mathcal{L}$. When a program state $s \in \Sigma$ satisfies the property φ, we write $s \models \varphi$ and we say that s is a φ-state. We also slightly abuse notation and write φ to also denote the set $\{s \in \Sigma \mid s \models \varphi\}$ of states that satisfy the property φ.

Example 9. Let us consider again the program SIMPLE from Figure 1. We write $\{\langle x,v\rangle\}$ to denote the environment $\rho\colon\{x\}\to\mathbb{Z}$ mapping the program variable x to the value $v\in\mathbb{Z}$. An example of state property allowed by the specification language that we have defined is the property $x=3$. The set of states that satisfy this property is $\{1,2,3,4,5,6,7\}\times\{\langle x,3\rangle\}$. Note however, that $\langle 7,\{\langle x,3\rangle\}\rangle$ is not reachable from the initial states. Another examples of state property allowed by the specification language is $7\colon x=3$, which is only satisfied by $\langle 7,\{\langle x,3\rangle\}\rangle$.■

The guarantee and recurrence properties within the hierarchy are then defined by means of the temporal operators *always* $\Box$ and *eventually* $\Diamond$.

5.1. Guarantee Properties

The class of guarantee properties is informally characterized as the class of properties stating that "something good happens *at least once*", that is, a program *eventually* reaches a desirable state. The guarantee properties that we consider are expressible by a temporal formula of the following form:

$$\Diamond\varphi$$

where φ is a state property. The temporal formula expresses that at least one program state in every program trace satisfies the property φ, but it does not promise any repetition. In general, these guarantee properties are used to ensure that some event happens once during a program execution.

A typical guarantee property is program *termination*, which ensures that all computations are finite, expressible by the temporal formula $\Diamond(l_e\colon\texttt{true})$, where $l_e\in\mathcal{L}$ denotes the program final control point.

Another typical guarantee property is program *total correctness*, which ensures that all computations starting in a φ-state terminate in a ψ-state, expressible by the temporal formula $\Diamond(l_i\colon\neg\varphi\vee l_e\colon\psi)$, where $l_i,l_e\in\mathcal{L}$ respectively denote the initial and final program control point.

Example 10. Let us consider again the program SIMPLE from Figure 1. An example of guarantee property is the formula $\Diamond(x=3)$, which is satisfied when the program initial states are limited to the set $\{\langle 1,\rho\rangle\in\Sigma\mid\rho(x)\leq 3\}$. In particular, note that when the initial states are limited to $\{\langle 1,\rho\rangle\in\Sigma\mid 0\leq\rho(x)\leq 3\}$, the guarantee property is satisfied within the first while loop. Instead, when the initial states are limited to $\{\langle 1,\rho\rangle\in\Sigma\mid\rho(x)<0\}$, the guarantee property is satisfied within the second while loop. Another example of guarantee property is $\Diamond(3\leq x)$, which is always satisfied by the program whatever its initial states.■

5.2. Recurrence Properties

The class of recurrence properties is informally characterized as the class of properties stating that "something good happens *infinitely often*", that is, a program reaches a desirable state *infinitely often*. The recurrence properties that we consider are expressible by a temporal formula of the following form:

$$\Box\Diamond\varphi$$

where φ is a state property. The temporal formula expresses that infinitely many program states in every program trace satisfy the property φ. In general, these recurrence properties are used to ensure that some event happens infinitely many times during a program execution.

A typical recurrence property is *starvation freedom*, which ensures that a process will repeatedly enter its critical section, and which is expressible by the temporal formula $\Box\Diamond(l_c\colon\texttt{true})$, where $l_c\in\mathcal{L}$ represents the critical section.

Example 11. Let us consider again the program SIMPLE from Figure 1. The recurrence property represented by the formula $\Box\Diamond x=3$ is satisfied when the program initial states are limited to the set $\{\langle 1,\rho\rangle\in\Sigma \mid \rho(x)<0\}$. In particular, note that the recurrence property is satisfied only within the second while loop. Instead, the recurrence property $\Box\Diamond 3\leq x$ is always satisfied by the program.■

6. Guarantee Semantics

In the following, we generalize Section 3 from termination to guarantee properties. We define a sound and complete semantics for proving guarantee temporal properties by abstract interpretation of the program maximal trace semantics. The generalization is straightforward but provides a building block for proving recurrence temporal properties in the next Section 7.

6.1. Fixpoint Guarantee Semantics

The *guarantee semantics*, given a set of desirable states $S\subseteq\Sigma$, is a ranking function $\tau_{\mathrm{g}}[S]\in\Sigma\rightharpoonup\mathbb{O}$ defined starting from the states in S, where the function has value zero, and retracing the program *backwards* while mapping every state in Σ definitely leading to a state in S (i.e., a state such that all the traces to which it belongs eventually reach a state in S) to an ordinal in $\mathbb{O}$ representing an upper bound on the number of program execution steps remaining to S. The domain $\mathrm{dom}(\tau_{\mathrm{g}}[S])$ of $\tau_{\mathrm{g}}[S]$ is the set of states definitely leading to a desirable state in S: all traces branching from a state $s\in\mathrm{dom}(\tau_{\mathrm{g}}[S])$ reach a state in S in at most $\tau_{\mathrm{g}}[S]s$ execution steps, while at least one trace branching from a state $s\notin\mathrm{dom}(\tau_{\mathrm{g}}[S])$ never reaches S.

Note that, the program traces that satisfy a guarantee property can also be *infinite* traces. In particular, guarantee properties are satisfied by finite subsequences of possibly infinite traces. Thus, in order to reason about subsequences, we define the function $\mathrm{sq}\colon\mathcal{P}(\Sigma^{+\infty})\to\mathcal{P}(\Sigma^{+})$, which extracts the finite subsequences of a set of sequences $T\subseteq\Sigma^{+\infty}$:

$$\mathrm{sq}(T)\stackrel{\text{def}}{=}\{\sigma\in\Sigma^{+}\mid\exists\sigma'\in\Sigma^{*},\sigma''\in\Sigma^{*\infty}\colon\sigma'\sigma\sigma''\in T\}\tag{17}$$

We recall that the neighborhood of a sequence $\sigma\in\Sigma^{+\infty}$ in a set of sequences $T\subseteq\Sigma^{+\infty}$ is the set of sequences $\sigma'\in T$ with a common prefix with σ (cf. Equation 5). A finite subsequence of a program trace satisfies a guarantee property if and only if it terminates in the desirable set of states (and never encounter a desirable state before), and its neighborhood in the subsequences of the program

semantics consists only of sequences that are terminating in the desirable set of states (and never encounter a desirable state before). The corresponding *guarantee abstraction* $\alpha^{\mathrm{g}}[S]\colon \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma^{+})$ is parameterized by a set of desirable states $S \subseteq \Sigma$ and it is defined as follows:

$$\alpha^{\mathrm{g}}[S]T \stackrel{\text{def}}{=} \left\{\sigma s \in \mathrm{sq}(T) \;\middle|\; \sigma \in \bar{S}^*,\ s \in S,\ \mathrm{nhbd}(\sigma, \mathrm{sf}(T) \cap \bar{S}^{+\infty}) = \emptyset\right\} \tag{18}$$

where $\bar{S} \stackrel{\text{def}}{=} \Sigma \setminus S$ and the function $\mathrm{sf}\colon \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma^{+\infty})$ yields the set of suffixes of a set of sequences $T \subseteq \Sigma^{+\infty}$:

$$\mathrm{sf}(T) \stackrel{\text{def}}{=} \bigcup \{\sigma \in \Sigma^{+\infty} \mid \exists \sigma' \in \Sigma^* : \sigma'\sigma \in T\}. \tag{19}$$

Example 12. Let $T \stackrel{\text{def}}{=} \{(abcd)^\omega, (cd)^\omega, a^\omega, cd^\omega\}$ and let $S \stackrel{\text{def}}{=} \{c\}$. We have $\mathrm{sf}(T) \cap \bar{S}^{+\infty} = \{a^\omega, d^\omega\}$. Then, we have $\alpha^{\mathrm{g}}[S]T = \{c, bc\}$. In fact, let us consider the trace $(abcd)^\omega$: the subsequences of $(abcd)^\omega$ that are terminating with c and never encounter c before are $\{c, bc, abc, dabc\}$; for abc, we have $\mathrm{pf}(ab) \cap \mathrm{pf}(a^\omega) = \{a\} \neq \emptyset$ and, for $dabc$, we have $\mathrm{pf}(dab) \cap \mathrm{pf}(d^\omega) = \{d\} \neq \emptyset$. Similarly, let us consider $(cd)^\omega$: the subsequences of $(cd)^\omega$ that are terminating with c and never encounter c before are $\{c, dc\}$; for dc, we have $\mathrm{pf}(d) \cap \mathrm{pf}(d^\omega) = \{d\} \neq \emptyset$.■

We can now define the guarantee semantics $\tau_{\mathrm{g}}[S] \in \Sigma \rightharpoonup \mathbb{O}$:

Definition 7 (Guarantee Semantics). Given a desirable set of states $S \subseteq \Sigma$, the *guarantee semantics* $\tau_{\mathrm{g}}[S] \in \Sigma \rightharpoonup \mathbb{O}$ is an abstract interpretation of the maximal trace semantics $\tau^{+\infty} \in \mathcal{P}(\Sigma^{+\infty})$ (cf. Equation 3):

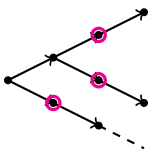
$$\tau_{\mathrm{g}}[S] \stackrel{\text{def}}{=} \alpha^{\mathrm{rk}}(\alpha^{\mathrm{g}}[S](\tau^{+\infty})) \tag{20}$$

where $\alpha^{\mathrm{rk}}\colon \mathcal{P}(\Sigma^{+}) \to (\Sigma \rightharpoonup \mathbb{O})$ is the ranking abstraction (cf. Equation 9).

The guarantee semantics can be expressed as a least fixpoint within the partially ordered set $\langle \Sigma \rightharpoonup \mathbb{O}, \sqsubseteq \rangle$ as follows:

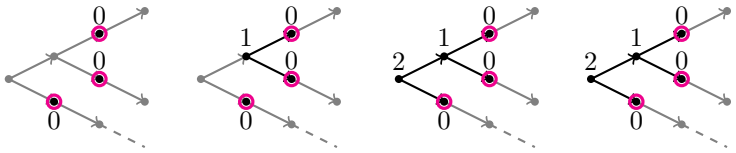
$$\begin{aligned} \tau_{\mathrm{g}}[S] &= \mathrm{lfp}^{\sqsubseteq}_{\dot{\emptyset}}\ \phi_{\mathrm{g}}[S] \\ \phi_{\mathrm{g}}[S]f &\stackrel{\text{def}}{=} \lambda s. \begin{cases} 0 & s \in S \\ \sup\{f(s')+1 \mid \langle s, s' \rangle \in \tau\} & s \notin S \wedge s \in \widetilde{\mathrm{pre}}(\mathrm{dom}(f)) \\ \text{undefined} & \text{otherwise} \end{cases} \end{aligned} \tag{21}$$

Example 13. Let us consider the following trace semantics:



where the highlighted states are the set S of desirable states.

The fixpoint iterates of the guarantee semantics $\tau_{\mathrm{g}}[S] \in \Sigma \rightharpoonup \mathbb{O}$ are:



where unlabelled states are outside the domain of the function. ■

Note that, when the set of desirable states S is the set of final states Ω, unsurprisingly we rediscover the termination semantics presented in Section 3, since $\phi_{\mathrm{g}}[\Omega] = \phi_{\mathrm{t}}$ (cf. Equation 4).

Let φ be a state property. The *φ-guarantee semantics* $\tau_{\mathrm{g}}^{\varphi} \in \Sigma \rightharpoonup \mathbb{O}$:

$$\tau_{\mathrm{g}}^{\varphi} \stackrel{\text{def}}{=} \tau_{\mathrm{g}}[\varphi] \tag{22}$$

is sound and complete for proving a guarantee property $\Diamond\varphi$:

Theorem 2. *A program satisfies a guarantee property $\Diamond\varphi$ for all traces starting from a given set of initial states $\mathcal{I}$ if and only if $\mathcal{I} \subseteq \mathrm{dom}(\tau_g^{\varphi})$.*

Proof. By Park's Fixpoint Induction Principle [15]. See Appendix B. □

6.2. Denotational Guarantee Semantics

In the following, we provide a structural definition of the fixpoint guarantee semantics $\tau_{\mathrm{g}}^{\varphi} \in \Sigma \rightharpoonup \mathbb{O}$ (cf. Equation 22) by induction on the syntax of programs written in our small imperative language presented in Section 4.

We partition $\tau_{\mathrm{g}}^{\varphi}$ with respect to the program control points: $\tau_{\mathrm{g}}^{\varphi} \in \mathcal{L} \to (\mathcal{E} \rightharpoonup \mathbb{O})$. In this way, to each program control point $l \in \mathcal{L}$ corresponds a partial function $f : \mathcal{E} \rightharpoonup \mathbb{O}$, and to each program instruction *stmt* corresponds a guarantee semantics transformer $\tau_{\mathrm{g}}^{\varphi}[\![\, stmt \,]\!] : (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$. Analogously to Equation 21, the ranking function is built *backwards*: each transformer $\tau_{\mathrm{g}}^{\varphi}[\![stmt]\!] : (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$ takes as input a ranking function whose domain represents the environments always leading to φ from the final control point of *stmt*, and determines the ranking function whose domain represents the environments always leading to φ from the initial control point of *stmt*, and whose value represents an upper bound on the number of program execution steps remaining to φ.

Skip Instruction. The guarantee semantics of a `skip` instruction *resets* the input ranking function $f : \mathcal{E} \rightharpoonup \mathbb{O}$ for the environments that satisfy φ, and otherwise it increases its value (as the `skip` termination semantics, cf. Equation 10):

$$\tau_{\mathrm{g}}^{\varphi}[\![\, {}^{l}\texttt{skip} \,]\!] f \stackrel{\text{def}}{=} \lambda\rho. \begin{cases} 0 & \langle l, \rho \rangle \models \varphi \\ f(\rho) + 1 & \langle l, \rho \rangle \not\models \varphi \wedge \rho \in \mathrm{dom}(f) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{23}$$

Assignment Instruction. Similarly, the guarantee semantics of a variable assignment ${}^{l}X := aexp$ *resets* the value of the input ranking function $f : \mathcal{E} \rightharpoonup \mathbb{O}$ for the environments that satisfy φ; otherwise, the resulting ranking function is defined over the environments that when subject to the variable assignment always belong to the domain of f (as the assignment termination semantics, cf. Equation 11):

$$\tau_{\mathrm{g}}^{\varphi}[\![{}^{l}X := aexp]\!]f \stackrel{\text{def}}{=} \lambda\rho. \begin{cases} 0 & \langle l,\rho\rangle \models \varphi \\ \sup\{f(\rho[X \leftarrow v]) + 1 \mid v \in [\![aexp]\!]\rho\} & \langle l,\rho\rangle \not\models \varphi \wedge \\ \qquad [\![aexp]\!]\rho \neq \emptyset \wedge \forall v' \in [\![aexp]\!]\rho : \rho[X \leftarrow v'] \in \mathrm{dom}(f) & \\ \text{undefined} & \text{otherwise} \end{cases} \tag{24}$$

Example 14. Let us consider again the program SIMPLE from Figure 1. We consider the following ranking function $f : \mathcal{E} \rightharpoonup \mathbb{O}$ valid at program point **3** during the first iterate of the guarantee semantics:

$$f \stackrel{\text{def}}{=} \lambda\rho. \begin{cases} 0 & \rho(x) = 3 \\ \text{undefined} & \text{otherwise} \end{cases}$$

the assignment $x := x+1$ at program point **5** and the guarantee property $\Diamond(x = 3)$. The guarantee semantics of the assignment, given the ranking function, is:

$$\tau_{\mathrm{g}}^{x=3}[\![x := x+1]\!]f \stackrel{\text{def}}{=} \lambda\rho. \begin{cases} 1 & \rho(x) = 2 \\ 0 & \rho(x) = 3 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note that the function is defined when $\rho(x) = 3$, even though $[\![x+1]\!]\rho = \{4\}$ and $\rho[x \leftarrow 4] \notin \mathrm{dom}(f)$. Indeed, the environment $\{\langle x,3\rangle\}$ satisfies the property $x = 3$.■

Conditional Instruction. Given a conditional `if` ${}^{l}bexp$ `then` $stmt_1$ `else` $stmt_2$ `fi`, its guarantee semantics takes as input a ranking function $f : \mathcal{E} \rightharpoonup \mathbb{O}$ and derives the guarantee semantics $\tau_{\mathrm{g}}^{\varphi}[\![stmt_1]\!]f$ and $\tau_{\mathrm{g}}^{\varphi}[\![stmt_2]\!]f$ of $stmt_1$, and $stmt_2$, respectively. Then, the guarantee semantics of the conditional instruction is defined by joining F_1, F_2, and F (defined exactly as for the `if` termination semantics, cf. Equation 12, where S_1 is $\tau_{\mathrm{g}}^{\varphi}[\![stmt_1]\!]f$ and S_2 is $\tau_{\mathrm{g}}^{\varphi}[\![stmt_2]\!]f$), and resetting the value of the function for the environments that satisfy φ:

$$\tau_{\mathrm{g}}^{\varphi}[\![\texttt{if } {}^{l}bexp \texttt{ then } stmt_1 \texttt{ else } stmt_2 \texttt{ fi}]\!]f \stackrel{\text{def}}{=} \lambda\rho. \begin{cases} 0 & \langle l,\rho\rangle \models \varphi \\ G(\rho) & \langle l,\rho\rangle \not\models \varphi \wedge \rho \in \mathrm{dom}(G) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{25}$$

where $G \stackrel{\text{def}}{=} F_1 \dot{\cup} F_2 \dot{\cup} F$.

Example 15. Let us consider again the program SIMPLE from Figure 1. We consider the guarantee property $\Diamond(x = 3)$ and the conditional statement

`if` *bexp* `then` *stmt*$_1$ `else` *stmt*$_2$ `fi` at program point 4. We assume, given $f : \mathcal{E} \rightharpoonup \mathbb{O}$ valid at program point 3 during some iterate, that the guarantee semantics of *stmt*$_1$ is defined as:

$$\tau_{\mathrm{g}}^{x=3}[\![\mathit{stmt}_1]\!]f \stackrel{\text{def}}{=} \lambda\rho. \begin{cases} 1 & \rho(x) \leq 0 \\ 0 & \rho(x) = 3 \\ \text{undefined} & \text{otherwise} \end{cases}$$

and that the guarantee semantics of *stmt*$_2$ is defined as

$$\tau_{\mathrm{g}}^{x=3}[\![\mathit{stmt}_2]\!]f \stackrel{\text{def}}{=} \lambda\rho. \begin{cases} 3 & 0 \leq \rho(x) < 3 \\ 0 & \rho(x) = 3 \\ 3 & 3 < \rho(x) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Then, since the boolean expression *bexp* is $x \leq 10$, the guarantee semantics of the conditional statement is:

$$\tau_{\mathrm{g}}^{x=3}[\![\texttt{if}\ ^{l}\mathit{bexp}\ \texttt{then}\ \mathit{stmt}_1\ \texttt{else}\ \mathit{stmt}_2\ \texttt{fi}]\!]f \stackrel{\text{def}}{=} \lambda\rho. \begin{cases} 2 & \rho(x) \leq 0 \\ 0 & \rho(x) = 3 \\ 4 & 3 < \rho(x) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Instead, if *bexp* is for example the non-deterministic choice `?`, we have:

$$\tau_{\mathrm{g}}^{x=3}[\![\texttt{if}\ ^{l}\mathit{bexp}\ \texttt{then}\ \mathit{stmt}_1\ \texttt{else}\ \mathit{stmt}_2\ \texttt{fi}]\!]f \stackrel{\text{def}}{=} \lambda\rho. \begin{cases} 4 & \rho(x) = 0 \\ 0 & \rho(x) = 3 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note that, unlike Example 8, both functions are also defined when $\rho(x) = 3$, since the environment $\{\langle x, 3\rangle\}$ satisfies the property $x = 3$. ■

Loop Instruction. The guarantee semantics of a loop `while` ^{l}*bexp* `do` *stmt* `od` takes as input a ranking function $f : \mathcal{E} \rightharpoonup \mathbb{O}$ whose domain represents the environments leading to φ from the final label of the instruction (i.e., after exiting the loop), and outputs the ranking function which is defined as the least fixpoint of the function $\phi_{\mathrm{g}}^{\varphi} : (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$ within $\langle \mathcal{E} \rightharpoonup \mathbb{O}, \sqsubseteq \rangle$, analogously to Equation 21:

$$\tau_{\mathrm{g}}^{\varphi}[\![\texttt{while}\ ^{l}\mathit{bexp}\ \texttt{do}\ \mathit{stmt}\ \texttt{od}]\!]f \stackrel{\text{def}}{=} \mathrm{lfp}_{\dot{\emptyset}}^{\dot{\sqsubseteq}}\ \phi_{\mathrm{g}}^{\varphi} \tag{26}$$

The function $\phi_{\mathrm{g}}^{\varphi} : (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$ takes as input a ranking function $x : \mathcal{E} \rightharpoonup \mathbb{O}$, resets its value for the environments that satisfy φ, and adds to its domain the environments for which one more loop iteration is needed before φ. The function $\phi_{\mathrm{g}}^{\varphi}$ is defined by joining the ranking functions F_1, F_2, and F (defined exactly as for the `while` termination semantics, cf. Equation 14, where S is the guarantee

semantics $\tau_{\mathrm{g}}^{\varphi}[\![stmt]\!]x$ of the loop body), and resetting the value of the function for the environments that satisfy φ:

$$\phi_{\mathrm{g}}^{\varphi}(x) \stackrel{\text{def}}{=} \lambda\rho.\begin{cases} 0 & \langle l,\rho\rangle \models \varphi \\ G(\rho) & \langle l,\rho\rangle \not\models \varphi \wedge \rho \in \mathrm{dom}(G) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{27}$$

where $G \stackrel{\text{def}}{=} F_1 \dot{\cup} F_2 \dot{\cup} F$.

Composition Instruction. Finally, the guarantee semantics of the sequential combination of instructions $stmt_1\ stmt_2$, takes as input a ranking function $f : \mathcal{E} \rightharpoonup \mathbb{O}$ at the final control point of $stmt_2$, determines from f the guarantee semantics $\tau_{\mathrm{g}}^{\varphi}[\![stmt_2]\!]f$ of $stmt_2$, and feeds it as input to the guarantee semantics of $stmt_1$ in order to get a ranking function at the initial control point of $stmt_1$:

$$\tau_{\mathrm{g}}^{\varphi}[\![stmt_1\ stmt_2]\!]f \stackrel{\text{def}}{=} \tau_{\mathrm{g}}^{\varphi}[\![stmt_1]\!](\tau_{\mathrm{g}}^{\varphi}[\![stmt_2]\!]f) \tag{28}$$

Program Guarantee Semantics. The guarantee semantics $\tau_{\mathrm{g}}^{\varphi}[\![prog]\!] \in \mathcal{E} \rightharpoonup \mathbb{O}$ of a program *prog* is a ranking function whose domain represents the environments eventually leading to φ, which is determined by taking as input the constant function equal to zero for the environments that satisfy φ, and undefined otherwise:

$$\tau_{\mathrm{g}}^{\varphi}[\![prog]\!] = \tau_{\mathrm{g}}^{\varphi}[\![stmt\ ^{l}]\!] \stackrel{\text{def}}{=} \tau_{\mathrm{g}}^{\varphi}[\![stmt]\!]\left(\lambda\rho.\begin{cases} 0 & \langle l,\rho\rangle \models \varphi \\ \text{undefined} & \text{otherwise} \end{cases}\right) \tag{29}$$

Note that, as pointed out in Remark 1, possible run-time errors are ignored. More specifically, all environments leading to run-time errors are discarded and do not belong to the domain of the guarantee semantics.

7. Recurrence Semantics

We now define a sound and complete semantics for proving recurrence temporal properties by abstract interpretation of the program maximal trace semantics, following the same approach used in Section 6 for guarantee properties.

7.1. Fixpoint Recurrence Semantics

The *recurrence semantics*, given a set of desirable states $S \subseteq \Sigma$, is a ranking function $\tau_{\mathrm{r}}[S] \in \Sigma \rightharpoonup \mathbb{O}$ defined starting from the states in S, where the function has value zero, and retracing the program *backwards* while mapping every state in Σ definitely leading *infinitely often* to a state in S (i.e., a state such that all the traces to which it belongs reach a state in S infinitely often) to an ordinal in $\mathbb{O}$ representing an upper bound on the number of program execution steps remaining to the next state in S. The domain $\mathrm{dom}(\tau_{\mathrm{r}}[S])$ of $\tau_{\mathrm{r}}[S]$ is the set of states definitely leading infinitely often to a desirable state in S: all traces branching from a state $s \in \mathrm{dom}(\tau_{\mathrm{r}}[S])$ reach the next state in S in at most $\tau_{\mathrm{r}}[S]s$

execution steps, while at least one trace branching from a state $s \notin \mathrm{dom}(\tau_{\mathrm{r}}[S])$ reaches S at most a finite number of times.

In particular, the recurrence semantics reuses the guarantee semantics of Section 6 as a building block: from the guarantee that some desirable event happens once during program execution, the recurrence semantics ensures that the event happens infinitely often. We define the set of subsequences of a program trace that satisfy a recurrence property using the set itself: a finite subsequence of a program trace satisfies a recurrence property if and only if it terminates in the desirable set of states (and never encounter a desirable state before), and its neighborhood in the subsequences of the program semantics consists only of sequences that are terminating in the desirable set of states (and never encounter a desirable state before), and that are *prefixes* of traces in the program semantics that reach infinitely often the desirable set of states. The corresponding *recurrence abstraction* $\alpha^{\mathrm{r}}[S] \colon \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma^{+})$ is parameterized by a set of desirable states $S \subseteq \Sigma$ and it is defined as a fixpoint as follows:

$$\begin{aligned} \alpha^{\mathrm{r}}[S]T &\stackrel{\text{def}}{=} \mathrm{gfp}^{\subseteq}_{\alpha^{\mathrm{g}}[S]T}\ \psi^{\mathrm{r}}[T,S] \\ \psi^{\mathrm{r}}[T,S]T' &\stackrel{\text{def}}{=} \alpha^{\mathrm{g}}[\widetilde{\mathrm{pre}}[T]T' \cap S]T \end{aligned} \tag{30}$$

where $\widetilde{\mathrm{pre}}[T]T' \stackrel{\text{def}}{=} \{s \in \Sigma \mid \forall \sigma \in \Sigma^*, \sigma' \in \Sigma^{*\infty} : \sigma s \sigma' \in T \Rightarrow \mathrm{pf}(\sigma') \cap T' \neq \emptyset\}$ is the set of states whose successors all belong to a given set of subsequences, and $\mathrm{gfp}_{\alpha^{\mathrm{g}}[S]}$ denotes the greatest fixpoint less than or equal to the guarantee abstraction $\alpha^{\mathrm{g}}[S] \colon \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma^{+})$ (cf. Equation 18) of T.

To explain intuitively Equation 30, we use the dual of Kleene's Fixpoint Theorem [11] to rephrase $\alpha^{\mathrm{r}}[S]$ as the following limit of a decreasing iteration:

$$\begin{aligned} \alpha^{\mathrm{r}}[S]T &= \bigcap_{i \in \mathbb{N}} T_{i+1} \\ T_{i+1} &\stackrel{\text{def}}{=} [\psi^{\mathrm{r}}[T,S]]^i(\alpha^{\mathrm{g}}[S]T) \end{aligned}$$

Then, for $i = 0$, we get the set $T_1 = \alpha^{\mathrm{g}}[S]T$ of subsequences of T that guarantee S at least *once*. For $i = 1$, starting from T_1, we derive the set of states $S_1 = \widetilde{\mathrm{pre}}[T]T_1 \cap S$ (i.e., $S_1 \subseteq S$) whose successors all belong to the subsequences in T_1, and we get the set $T_2 = \alpha^{\mathrm{g}}[S_1]T$ of subsequences of T that guarantee S_1 at least once and thus guarantee S at least *twice*. Note that all the subsequences in T_2 terminate with a state $s' \in S_1$ and therefore are *prefixes* of subsequence of T that reach S at least twice. More generally, for each $i \in \mathbb{N}$, we get the set T_{i+1} of subsequences which are *prefixes* of subsequences of T that reach S at least $i+1$ times, i.e., the subsequences that guarantee S at least $i+1$ times. The limit thus guarantees S *infinitely often*.

Example 16. Let $T \stackrel{\text{def}}{=} \{(cd)^\omega, ca^\omega, d(be)^\omega\}$ and let $S \stackrel{\text{def}}{=} \{b, c, d\}$. For $i = 0$, we have $T_1 = \alpha^{\mathrm{g}}[S]T = \{b, eb, c, d\}$. For $i = 1$, we derive $S_1 = \{b, d\}$, since $c(dc)^\omega \in T$ and $\mathrm{pf}((dc)^\omega) \cap T_1 = \{d\} \neq \emptyset$ but $ca^\omega \in T$ and $\mathrm{pf}(a^\omega) \cap T_1 = \emptyset$. We get $T_2 = \alpha^{\mathrm{g}}[S_1]T = \{b, eb, d\}$. For $i = 2$, we derive $S_2 = \{b\}$, since $d(be)^\omega \in T$

and $\mathrm{pf}((be)^\omega)\cap T_1=\{b\}\neq\emptyset$ but $d(cd)^\omega\in T$ and $\mathrm{pf}((cd)^\omega)\cap T_2=\emptyset$. We get $T_3=\alpha^{\mathrm{g}}[S_2]T=\{b,eb\}$ which is the greatest fixpoint: the only subsequences of sequences in T that guarantee S infinitely often start with b or eb. ∎

We can now define the recurrence semantics $\tau_{\mathrm{r}}[S]\in\Sigma\rightharpoonup\mathbb{O}$:

Definition 8 (Recurrence Semantics). Given a desirable set of states $S\subseteq\Sigma$, the *recurrence semantics* $\tau_{\mathrm{r}}[S]\in\Sigma\rightharpoonup\mathbb{O}$ is an abstract interpretation of the maximal trace semantics $\tau^{+\infty}\in\mathcal{P}(\Sigma^{+\infty})$ (cf. Equation 3):

$$\tau_{\mathrm{r}}[S]\stackrel{\text{def}}{=}\alpha^{\mathrm{rk}}(\alpha^{\mathrm{r}}[S](\tau^{+\infty})) \tag{31}$$

where $\alpha^{\mathrm{rk}}\colon\mathcal{P}(\Sigma^+)\to(\Sigma\rightharpoonup\mathbb{O})$ is the ranking abstraction (cf. Equation 9).

The recurrence semantics can be expressed as a least fixpoint within the partially ordered set $\langle\Sigma\rightharpoonup\mathbb{O},\sqsubseteq\rangle$ as follows:

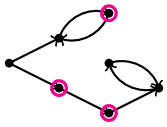
$$\begin{aligned}\tau_{\mathrm{r}}[S]&=\mathrm{gfp}^{\sqsubseteq}_{\tau_{\mathrm{g}}[S]}\ \phi_{\mathrm{r}}[S]\\ \phi_{\mathrm{r}}[S]f&\stackrel{\text{def}}{=}\lambda s.\begin{cases}f(s) & s\in\mathrm{dom}(\tau_{\mathrm{g}}[\widetilde{\mathrm{pre}}(\mathrm{dom}(f))\cap S])\\ \text{undefined} & \text{otherwise}\end{cases}\end{aligned} \tag{32}$$

Note that, the recurrence semantics can be equivalently simplified as:

$$\begin{aligned}\tau_{\mathrm{r}}[S]&=\mathrm{gfp}^{\sqsubseteq}_{\tau_{\mathrm{g}}[S]}\ \phi_{\mathrm{r}}[S]\\ \phi_{\mathrm{r}}[S]f&\stackrel{\text{def}}{=}\lambda s.\begin{cases}f(s) & s\in\widetilde{\mathrm{pre}}(\mathrm{dom}(f))\\ \text{undefined} & \text{otherwise}\end{cases}\end{aligned} \tag{33}$$

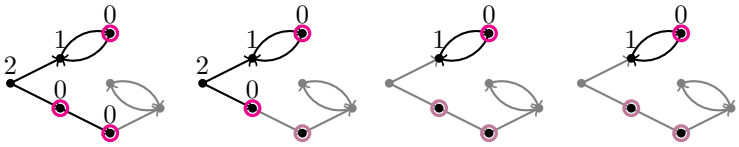
Indeed, there is not need to redefine $\tau_{\mathrm{g}}[S]$ at each iterate since we always have $\mathrm{dom}(f)\subseteq\mathrm{dom}(\tau_{\mathrm{g}}[S])$ and $\forall s\in\mathrm{dom}(f)\colon f(s)=\tau_{\mathrm{g}}[S](s)$.

Example 17. Let us consider the following trace semantics:



where the highlighted states are the set S of desirable states.

The fixpoint iterates of the recurrence semantics $\tau_{\mathrm{r}}[S]\in\Sigma\rightharpoonup\mathbb{O}$ are:



where unlabelled states are outside the domain of the function. ∎

Let φ be a state property. The *φ-recurrence semantics* $\tau_{\mathrm{r}}^{\varphi} \in \Sigma \rightharpoonup \mathbb{O}$:

$$\tau_{\mathrm{r}}^{\varphi} \stackrel{\text{def}}{=} \tau_{\mathrm{r}}[\varphi] \tag{34}$$

is sound and complete for proving a recurrence property $\Box\Diamond\varphi$:

Theorem 3. *A program satisfies a recurrence property $\Box\Diamond\varphi$ for all traces starting from a given set of states $\mathcal{I}$ if and only if $\mathcal{I} \subseteq \mathrm{dom}(\tau_r^{\varphi})$.*

Proof. The proof follows from the dual of Park's Fixpoint Induction Principle [15] and from Theorem 2. See Appendix B. □

7.2. Denotational Recurrence Semantics

In the following, we provide a structural definition of the fixpoint recurrence semantics $\tau_{\mathrm{r}}^{\varphi} \in \Sigma \rightharpoonup \mathbb{O}$ (cf. Equation 34) by induction on the syntax of programs written in our idealized programming language of Section 4.

We partition $\tau_{\mathrm{r}}^{\varphi}$ with respect to the program control points: $\tau_{\mathrm{r}}^{\varphi} \in \mathcal{L} \to (\mathcal{E} \rightharpoonup \mathbb{O})$. In this way, to each program control point $l \in \mathcal{L}$ corresponds a partial function $f : \mathcal{E} \rightharpoonup \mathbb{O}$, and to each program instruction *stmt* corresponds a recurrence semantics transformer $\tau_{\mathrm{r}}^{\varphi}[\![stmt]\!] : (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$. Analogously to Equation 33, the ranking function is built *backwards*: each transformer $\tau_{\mathrm{r}}^{\varphi}[\![stmt]\!] : (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$ takes as input a ranking function whose domain represents the environments always leading *infinitely often* to φ from the final control point of *stmt*, and determines a ranking function whose domain represents the environments always leading *infinitely often* to φ from the initial control point of *stmt*, and whose value represents an upper bound on the number of program execution steps remaining to the next occurrence of φ. In particular, each transformer $\tau_{\mathrm{r}}^{\varphi}[\![stmt]\!] \in (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$ behaves as the guarantee semantics transformer $\tau_{\mathrm{g}}^{\varphi}[\![stmt]\!] \in (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$ defined in Section 6.2 and also ensures that each time φ is satisfied, it will be satisfied again in the future: the value of the input ranking function is reset for the environments that satisfy φ only if all their successors by means of the instruction *stmt* belong to the domain of the input ranking function.

Skip Instruction. The recurrence semantics of a `skip` instruction is defined analogously to its guarantee semantics (cf. Equation 23), except that it resets the value of the input ranking function $f : \mathcal{E} \rightharpoonup \mathbb{O}$ for the environments that satisfy φ only when they already belong to its domain:

$$\tau_{\mathrm{r}}^{\varphi}[\![^{l}\texttt{skip}]\!] f \stackrel{\text{def}}{=} \lambda\rho. \begin{cases} 0 & \langle l,\rho\rangle \models \varphi \wedge \rho \in \mathrm{dom}(f) \\ f(\rho)+1 & \langle l,\rho\rangle \not\models \varphi \wedge \rho \in \mathrm{dom}(f) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{35}$$

Assignment Instruction. Similarly, the recurrence semantics of a variable assignment ${}^{l}X := aexp$ is defined analogously to the assignment guarantee semantics (cf. Equation 24), except that it resets the value of the input ranking function $f : \mathcal{E} \rightharpoonup \mathbb{O}$ only for the environments that satisfy φ and that when subject to the assignment always belong to the domain of f:

$$\tau_{\mathrm{r}}^{\varphi}[\![{}^{l}X := aexp]\!]f \stackrel{\text{def}}{=} \lambda\rho. \begin{cases} 0 & \langle l,\rho\rangle \models \varphi \wedge \\ & \quad [\![aexp]\!]\rho \neq \emptyset \wedge \forall v' \in [\![aexp]\!]\rho : \rho[X \leftarrow v'] \in \mathrm{dom}(f) \\ \sup\{f(\rho[X \leftarrow v]) + 1 \mid v \in [\![aexp]\!]\rho\} & \langle l,\rho\rangle \not\models \varphi \wedge \\ & \quad [\![aexp]\!]\rho \neq \emptyset \wedge \forall v' \in [\![aexp]\!]\rho : \rho[X \leftarrow v'] \in \mathrm{dom}(f) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{36}$$

Example 18. Let us consider again the program SIMPLE from Figure 1. We consider the following ranking function $f : \mathcal{E} \rightharpoonup \mathbb{O}$ valid at program point **3** during the first iterate of the recurrence semantics:

$$f \stackrel{\text{def}}{=} \lambda\rho. \begin{cases} 0 & \rho(x) = 3 \\ \text{undefined} & \text{otherwise} \end{cases}$$

the assignment $x := x + 1$ at program point **5** and the recurrence property $\Box\Diamond(x = 3)$. The recurrence semantics of the assignment is:

$$\tau_{\mathrm{r}}^{x=3}[\![x := x+1]\!]f \stackrel{\text{def}}{=} \lambda\rho. \begin{cases} 1 & \rho(x) = 2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note that, unlike Example 14, the function is not defined when $\rho(x) = 3$, since $\{\langle x,3\rangle\}$ satisfies the property $x = 3$ but $[\![x+1]\!]\rho = \{4\}$ and $\rho[x \leftarrow 4] \notin \mathrm{dom}(f)$. ■

Conditional Instruction. The recurrence semantics of a conditional instruction `if` ${}^{l}bexp$ `then` $stmt_1$ `else` $stmt_2$ `fi`, unlike its guarantee semantics (cf. Equation 25), resets the value of the function obtained by joining F_1, F_2, and F (cf. Equation 12, where S_1 is $\tau_{\mathrm{r}}^{\varphi}[\![stmt_1]\!]f$ and S_2 is $\tau_{\mathrm{r}}^{\varphi}[\![stmt_2]\!]f$) only for the environments that satisfy φ and also belong to its domain:

$$\tau_{\mathrm{g}}^{\varphi}[\![\texttt{if } {}^{l}bexp \texttt{ then } stmt_1 \texttt{ else } stmt_2 \texttt{ fi}]\!]f \stackrel{\text{def}}{=} \lambda\rho. \begin{cases} 0 & \langle l,\rho\rangle \models \varphi \wedge \rho \in \mathrm{dom}(R) \\ R(\rho) & \langle l,\rho\rangle \not\models \varphi \wedge \rho \in \mathrm{dom}(R) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{37}$$

where $R \stackrel{\text{def}}{=} F_1 \dot{\cup} F_2 \dot{\cup} F$.

Example 19. Let us consider again the program SIMPLE from Figure 1. We consider the recurrence property $\Box\Diamond(x = 3)$ and the conditional statement `if` $bexp$ `then` $stmt_1$ `else` $stmt_2$ `fi` at program point **4**. We assume, given

$f : \mathcal{E} \rightharpoonup \mathbb{O}$ valid at program point 3 during some iterate, that the recurrence semantics of $stmt_1$ is defined as:

$$\tau_{\mathrm{r}}^{x=3}[\![stmt_1]\!]f \stackrel{\text{def}}{=} \lambda\rho. \begin{cases} 1 & \rho(x) \leq 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

and that the recurrence semantics of $stmt_2$ is defined as

$$\tau_{\mathrm{r}}^{x=3}[\![stmt_2]\!]f \stackrel{\text{def}}{=} \lambda\rho. \begin{cases} 3 & 0 \leq \rho(x) < 3 \\ 0 & \rho(x) = 3 \\ 3 & 3 < \rho(x) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Then, since the boolean expression $bexp$ is $x \leq 10$, the recurrence semantics of the conditional statement is:

$$\tau_{\mathrm{r}}^{x=3}[\![\texttt{if } {}^{l} bexp \texttt{ then } stmt_1 \texttt{ else } stmt_2 \texttt{ fi}]\!]f \stackrel{\text{def}}{=} \lambda\rho. \begin{cases} 2 & \rho(x) \leq 0 \\ 4 & 3 < \rho(x) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Instead, if $bexp$ is for example the non-deterministic choice `?`, we have:

$$\tau_{\mathrm{r}}^{x=3}[\![\texttt{if } {}^{l} bexp \texttt{ then } stmt_1 \texttt{ else } stmt_2 \texttt{ fi}]\!]f \stackrel{\text{def}}{=} \lambda\rho. \begin{cases} 4 & \rho(x) = 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note that, unlike Example 15, both functions are undefined when $\rho(x) = 3$, even though the property $x = 3$ is satisfied by the environment $\{\langle x, 3\rangle\}$. In fact, the ranking function for the `then` branch of the `if` is undefined when $\rho(x) = 3$. ■

Loop Instruction. The recurrence semantics of a loop $\texttt{while } {}^{l} bexp \texttt{ do } stmt \texttt{ od}$ takes as input a ranking function $f : \mathcal{E} \rightharpoonup \mathbb{O}$ whose domain represents the environments leading infinitely often to φ from the final label of the instruction (i.e., after exiting the loop), and outputs the ranking function which is defined as a greatest fixpoint of the function $\phi_{\mathrm{r}}^{\varphi} : (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$ within $\langle \mathcal{E} \rightharpoonup \mathbb{O}, \sqsubseteq \rangle$:

$$\tau_{\mathrm{r}}^{\varphi}[\![\texttt{while } {}^{l} bexp \texttt{ do } stmt \texttt{ od}]\!]f \stackrel{\text{def}}{=} \mathrm{gfp}_{G}^{\sqsubseteq}\, \phi_{\mathrm{r}}^{\varphi} \tag{38}$$

where $G \stackrel{\text{def}}{=} \tau_{\mathrm{g}}^{\varphi}[\![\texttt{while } {}^{l} bexp \texttt{ do } stmt \texttt{ od}]\!]f$ is the guarantee semantics of the loop instruction defined in Equation 26. In essence, from the guarantee that some desirable event eventually happens, the recurrence semantics ensures that the event happens infinitely often. The function $\phi_{\mathrm{r}}^{\varphi} : (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$ takes as input a ranking function $x : \mathcal{E} \rightharpoonup \mathbb{O}$, resets its value for the environments that belong to its domain and that satisfy φ, and adds to its domain the environments for which one more loop iteration is needed before the next occurrence of φ. The function $\phi_{\mathrm{r}}^{\varphi}$, unlike $\phi_{\mathrm{g}}^{\varphi}$ (cf. Equation 27), resets the value of the function

obtained by joining F_1, F_2, and F (cf. Equation 14, where S is $\tau_{\mathrm{r}}^{\varphi}[\![stmt]\!]x$) only for the environments that satisfy φ and also belong to its domain:

$$\phi_{\mathrm{r}}^{\varphi}(x) \stackrel{\text{def}}{=} \lambda\rho.\begin{cases} 0 & \langle l,\rho\rangle \models \varphi \wedge \rho \in \mathrm{dom}(R) \\ R(\rho) & \langle l,\rho\rangle \models \varphi \wedge \rho \in \mathrm{dom}(R) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{39}$$

where $R \stackrel{\text{def}}{=} F_1 \dot{\cup} F_2 \dot{\cup} F$.

Composition Instruction. Finally, the recurrence semantics of the sequential combination of instructions $stmt_1$ $stmt_2$, takes as input a ranking function $f : \mathcal{E} \rightharpoonup \mathbb{O}$ at the final control point of $stmt_2$, determines from f the recurrence semantics $\tau_{\mathrm{r}}^{\varphi}[\![stmt_2]\!]f$ of $stmt_2$, and feeds it as input to the recurrence semantics of $stmt_1$ in order to get a ranking function at the initial control point of $stmt_1$:

$$\tau_{\mathrm{r}}^{\varphi}[\![stmt_1\ stmt_2]\!]f \stackrel{\text{def}}{=} \tau_{\mathrm{r}}^{\varphi}[\![stmt_1]\!](\tau_{\mathrm{r}}^{\varphi}[\![stmt_2]\!]f) \tag{40}$$

Program Recurrence Semantics. The recurrence semantics $\tau_{\mathrm{r}}^{\varphi}[\![prog]\!] \in \mathcal{E} \rightharpoonup \mathbb{O}$ of a program *prog* is a ranking function whose domain represents the environments leading *infinitely often* to φ, which is determined by taking as input the totally undefined function, since the program final states cannot satisfy a recurrence property:

$$\tau_{\mathrm{r}}^{\varphi}[\![prog]\!] = \tau_{\mathrm{r}}^{\varphi}[\![stmt\ ^{l}]\!] \stackrel{\text{def}}{=} \tau_{\mathrm{r}}^{\varphi}[\![stmt]\!]\dot{\emptyset} \tag{41}$$

As pointed out in Remark 1, possible run-time errors are ignored. Thus, all environments leading to run-time errors are discarded and do not belong to the domain of the recurrence semantics.

8. Piecewise-Defined Ranking Functions

The termination semantics τ_{t} of Section 3, the φ-guarantee semantics $\tau_{\mathrm{g}}^{\varphi}$ of Section 6 and the φ-recurrence semantics $\tau_{\mathrm{r}}^{\varphi}$ of Section 7 are usually *not computable* (i.e., when the program state space is infinite).

In [6, 7, 8], we present decidable abstractions of τ_{t} by means of *piecewise-defined* ranking functions over natural numbers [6], over ordinals [7] and with relational partitioning [8]. In the following, we will briefly recall the main characteristics of these abstractions and we will show how to modify the abstract domains in order to obtain decidable abstractions of $\tau_{\mathrm{g}}^{\varphi}$ and $\tau_{\mathrm{r}}^{\varphi}$ as well. We refer to [6, 7, 8] for more detailed discussions on the abstract domains.

8.1. Abstract Termination Semantics

The abstract termination semantics $\tau_{\mathrm{t}}^{\sharp} \in \mathcal{L} \to \mathcal{T}$ maps each program control point $l \in \mathcal{L}$ to an element $t \in \mathcal{T}$ of the decision trees abstract domain $\mathcal{T}$.

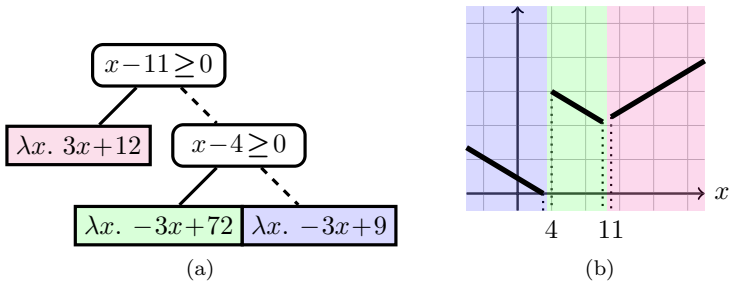


Figure 5: Decision tree representation (a) of the piecewise-defined ranking function (b) inferred for proving $\Diamond(x=3)$ and $\Box\Diamond(x=3)$ at program control point **3** of the program SIMPLE of Figure 1 . The linear constraints within the decision nodes are satisfied by their left subtree, while their right subtree satisfies their negation. The leaves of the decision tree represent partial functions whose domain is determined by the constraints satisfied along the path to the leaf node.

Decision Trees Abstract Domain. The elements of the decision trees abstract domain $\mathcal{T}$ are piecewise-defined ranking functions represented by *decision trees*, where the decision nodes are labeled with linear constraints in $\mathcal{C} \stackrel{\text{def}}{=} \{c_1X_1+\cdots+c_kX_k+c_{k+1}\geq 0 \mid X_1,\ldots,X_k\in\mathcal{X},\ c_1,\ldots,c_k,c_{k+1}\in\mathbb{Z}\}$, and the leaf nodes belong to an auxiliary abstract domain $\mathcal{F}$ whose elements are natural-valued (or ordinal-valued [7]) functions of the program variables. The paths along the decision trees establish the shape of the pieces of the ranking functions, and the leaf nodes represent the value of the ranking functions within their pieces. A special element $\bot$ denotes an undefined value within a piece. In the following, we slightly abuse notation and use $\bot$ to also denote a decision tree with a single undefined leaf node.

The decision trees abstract domain is parametric in the choice between the expressivity and the cost of the numerical abstract domain [16, 17, 18] which underlies the linear constraints labeling the decision nodes, and the choice of the auxiliary abstract domain for the leaf nodes. For example, in [6] we consider piecewise-defined ranking functions represented using interval constraints based on the intervals abstract domain [16] at the decision nodes, and affine functions at the leaf nodes. We used the same parameterization to analyze the program SIMPLE of Figure 1 for proving $\Diamond(x=3)$ and $\Box\Diamond(x=3)$ and, in Figure 5a, we depict the decision tree inferred at program control point **3**. The graphical representation of the ranking function is shown in Figure 5b.

Abstract Termination Semantics. A *sound* abstract termination semantics transformer $\tau_{\mathrm{t}}^{\natural}[\![stmt]\!]\in\mathcal{T}\to\mathcal{T}$ corresponds to each program instruction *stmt*. We define each function $\tau_{\mathrm{t}}^{\natural}[\![stmt]\!]$ in Figure 7 by means of the following operators in the decision trees abstract domain: STEP, B-ASSIGN$[\![X:=aexp]\!]$, FILTER$[\![bexp]\!]$, the join operator $\curlyvee$, and the widening operator $\triangledown$. The operator STEP descends along each path of a decision tree up to a leaf node, where it simply increments the value of the ranking function (cf. Figure 6b) to count another program execution step. The operator B-ASSIGN$[\![X:=aexp]\!]$ models a backward assignment by substituting the variable X with the expression *aexp* within the decision nodes as well as

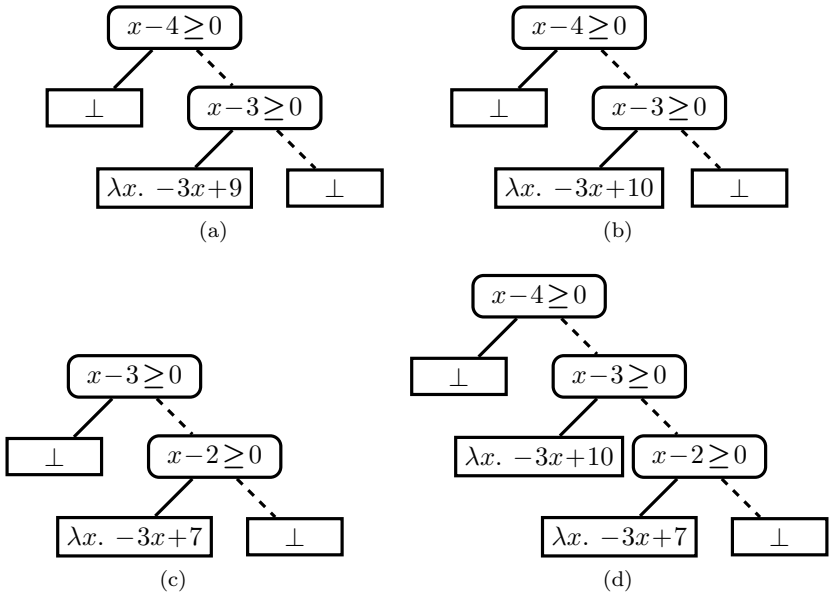


Figure 6: The decision tree (a) is obtained by the FILTER$[\![x=3]\!]$ operator from Figure 5a. The decision tree (b) is obtained by the STEP operator from (a). The decision tree (c) is the result of the B-ASSIGN$[\![x:=x+1]\!]$ operator on (a). The decision tree (d) is the join of (b) and (c).

within the leaf nodes, and also increments the value of the ranking function within the leaf nodes (cf. Figure 6c). The operator FILTER$[\![bexp]\!]$ discards all paths of a decision tree that do not satisfy the expression *bexp*, possibly introducing new decision nodes, and also increments the value of the ranking function within the remaining leaf nodes (cf. Figure 6a). The join operator yields a piecewise-defined ranking function defined over the coarsest partition refining both partitions of the given decision trees (cf. Figure 6d). The widening operator instead imposes the less refined partition of a given decision tree upon another given decision tree, possibly inducing a loss of precision but enforcing termination of the analysis. In Figure 7, $\mathrm{lfp}^\natural\ \phi_{\mathrm{t}}^\natural$ denotes the limit of the iteration sequence with widening:

$$\begin{aligned} y_0 &\stackrel{\text{def}}{=} \bot \\ y_{n+1} &\stackrel{\text{def}}{=} \begin{cases} y_n & \phi_{\mathrm{t}}^\natural(y_n) \sqsubseteq^\natural y_n \wedge \phi_{\mathrm{t}}^\natural(y_n) \preccurlyeq^\natural y_n \\ y_n \triangledown \phi_{\mathrm{t}}^\natural(y_n) & \text{otherwise} \end{cases} \end{aligned} \tag{42}$$

where $\sqsubseteq^\natural$ and $\preccurlyeq^\natural$ are the abstract counterparts of the computational $\sqsubseteq$ and approximation $\preccurlyeq$ order, respectively. We refer to [6, 7, 8] for further details.

The transformers $\tau_{\mathrm{t}}^\natural[\![stmt]\!]$ are combined together to compute a piecewise-defined ranking function for a program through *backward* analysis. The starting point is the constant function equal to zero at the program final control point $f[\![prog]\!]$. This function is then propagated backwards towards the program initial

$$
\begin{aligned}
&\tau_{\mathrm{t}}^{\natural}[\![{}^{l}\texttt{skip}]\!]t \stackrel{\text{def}}{=} \text{STEP}(t)\\
&\tau_{\mathrm{t}}^{\natural}[\![{}^{l}X := aexp]\!]t \stackrel{\text{def}}{=} \text{B-ASSIGN}[\![X := aexp]\!]t\\
&\tau_{\mathrm{t}}^{\natural}[\![\texttt{if } {}^{l}bexp \texttt{ then } stmt_1 \texttt{ else } stmt_2 \texttt{ fi}]\!]t \stackrel{\text{def}}{=}\\
&\qquad \text{FILTER}[\![bexp]\!](\tau_{\mathrm{t}}^{\natural}[\![stmt_1]\!]t) \curlyvee \text{FILTER}[\![\texttt{not } bexp]\!](\tau_{\mathrm{t}}^{\natural}[\![stmt_2]\!]t)\\
&\tau_{\mathrm{t}}^{\natural}[\![\texttt{while } {}^{l}bexp \texttt{ do } stmt \texttt{ od}]\!]t \stackrel{\text{def}}{=} \text{lfp}^{\natural}\ \phi_{\mathrm{t}}^{\natural}\\
&\qquad \phi_{\mathrm{t}}^{\natural}(x) \stackrel{\text{def}}{=} \text{FILTER}[\![bexp]\!](\tau_{\mathrm{t}}^{\natural}[\![stmt]\!]x) \curlyvee \text{FILTER}[\![\texttt{not } bexp]\!]t\\
&\tau_{\mathrm{t}}^{\natural}[\![stmt_1\ stmt_2]\!]t \stackrel{\text{def}}{=} \tau_{\mathrm{t}}^{\natural}[\![stmt_1]\!](\tau_{\mathrm{t}}^{\natural}[\![stmt_2]\!]t)
\end{aligned}
$$

Figure 7: Abstract termination semantics of instructions *stmt*.

$$
\begin{aligned}
&\tau_{\mathrm{g}}^{\varphi\natural}[\![{}^{l}\texttt{skip}]\!]t \stackrel{\text{def}}{=} \text{RESET}[\![\varphi]\!](\text{STEP}(t))\\
&\tau_{\mathrm{g}}^{\varphi\natural}[\![{}^{l}X := aexp]\!]t \stackrel{\text{def}}{=} \text{RESET}[\![\varphi]\!](\text{B-ASSIGN}[\![X := aexp]\!]t)\\
&\tau_{\mathrm{g}}^{\varphi\natural}[\![\texttt{if } {}^{l}bexp \texttt{ then } stmt_1 \texttt{ else } stmt_2 \texttt{ fi}]\!]t \stackrel{\text{def}}{=}\\
&\qquad \text{RESET}[\![\varphi]\!](\text{FILTER}[\![bexp]\!](\tau_{\mathrm{g}}^{\varphi\natural}[\![stmt_1]\!]t) \curlyvee \text{FILTER}[\![\texttt{not } bexp]\!](\tau_{\mathrm{g}}^{\varphi\natural}[\![stmt_2]\!]t))\\
&\tau_{\mathrm{g}}^{\varphi\natural}[\![\texttt{while } {}^{l}bexp \texttt{ do } stmt \texttt{ od}]\!]t \stackrel{\text{def}}{=} \text{lfp}^{\natural}\ \phi_{\mathrm{g}}^{\varphi\natural}\\
&\qquad \phi_{\mathrm{g}}^{\varphi\natural}(x) \stackrel{\text{def}}{=} \text{RESET}[\![\varphi]\!](\text{FILTER}[\![bexp]\!](\tau_{\mathrm{g}}^{\varphi\natural}[\![stmt]\!]x) \curlyvee \text{FILTER}[\![\texttt{not } bexp]\!]t)\\
&\tau_{\mathrm{g}}^{\varphi\natural}[\![stmt_1\ stmt_2]\!]t \stackrel{\text{def}}{=} \tau_{\mathrm{g}}^{\varphi\natural}[\![stmt_1]\!](\tau_{\mathrm{g}}^{\varphi\natural}[\![stmt_2]\!]t)
\end{aligned}
$$

Figure 8: Abstract guarantee semantics of instructions *stmt*.

control point $i[\![prog]\!]$ taking assignments and tests into account and, in case of loops, solving least fixpoints by iteration with widening.

The abstract termination semantics is *sound* with respect to the approximation order $v_1 \preccurlyeq v_2 \Leftrightarrow \mathrm{dom}(v_1) \supseteq \mathrm{dom}(v_2) \wedge \forall x \in \mathrm{dom}(v_2) : v_1(x) \leq v_2(x)$ (cf. Section 3). Thus, the backward analysis computes an over-approximation of the value of the termination semantics τ_{t} and an *under-approximation* of its domain of definition $\mathrm{dom}(\tau_{\mathrm{t}})$. In this way, an abstraction provides *sufficient preconditions* for program termination: if the abstraction is defined on a program state, then all the program traces branching from that state are terminating, and the value of the function provides an upper bound on the number of execution steps before termination.

8.2. Abstract Guarantee Semantics

In the following, we describe how to reuse the decision trees abstract domain [6, 7, 8] briefly recalled in the previous section, and what changes are required in order to obtain decidable abstractions of $\tau_{\mathrm{g}}^{\varphi}$ (cf. Equation 22).

We define the abstract φ-guarantee semantics $\tau_{\mathrm{g}}^{\varphi\natural} \in \mathcal{L} \to \mathcal{T}$: to each program control point $l \in \mathcal{L}$ corresponds a piecewise-defined ranking function $t \in \mathcal{T}$, and for each program instruction *stmt* a *sound* guarantee semantics transformer $\tau_{\mathrm{g}}^{\varphi\natural}[\![stmt]\!] \in \mathcal{T} \to \mathcal{T}$ is defined in Figure 8. In particular, we complement the operators briefly presented in the previous Section 8.1 with a new operator $\text{RESET}[\![\varphi]\!]$, which possibly splits a given piecewise-defined ranking function into more pieces (by introducing new decision nodes in a decision tree) in order to distinguish the

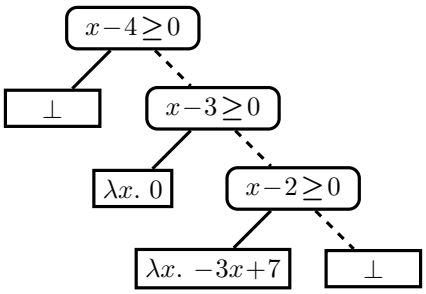


Figure 9: Decision tree obtained by the RESET$[\![x=3]\!]$ operator from Figure 6c.

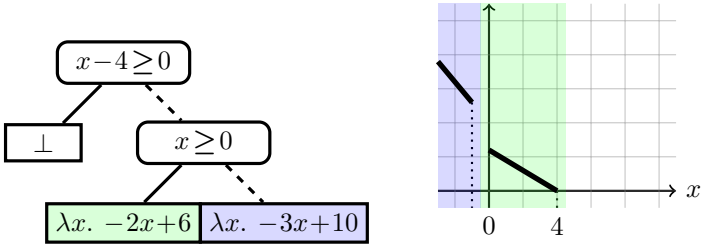


Figure 10: Decision tree representation (a) of the piecewise-defined ranking function (b) inferred for proving $\Diamond(x=3)$ at program control point 1 of Figure 1.

pieces that satisfy φ, and resets its value within those pieces (and leaves the other pieces unchanged). We propose an example of use of the RESET$[\![\varphi]\!]$ operator in Figure 9. Note that, RESET$[\![\varphi]\!]$ operates also on undefined leaf nodes.

The transformers $\tau_{\mathrm{g}}^{\varphi\natural}[\![stmt]\!]$ are again combined together through *backward* analysis. The starting point is now the constant function equal to zero only for the environments that satisfy the property φ, and undefined elsewhere (i.e., RESET$[\![\varphi]\!]\bot$), at the program final control point $f[\![prog]\!]$. The backward analysis computes an over-approximation of the value of the φ-guarantee semantics $\tau_{\mathrm{g}}^{\varphi}$ and an *under-approximation* of its domain of definition $\mathrm{dom}(\tau_{\mathrm{g}}^{\varphi})$. In this way, an abstraction provides *sufficient preconditions* for the guarantee property $\Diamond\varphi$: if the abstraction is defined on a program state, then all the program traces branching from that state *eventually* reach a state with the property φ, and the value of the function provides an upper bound on the number of execution steps before such state with the property φ.

Example 20. In Figure 10a, we depict the decision tree inferred for proving the guarantee property $\Diamond(x=3)$ at program control point 1 of the program SIMPLE of Figure 1. The graphical representation of the ranking function is shown in Figure 10b. Its domain yields the sufficient precondition $x\leq 3$ for $\Diamond(x=3)$. ■

$\tau_{\mathrm{r}}^{\varphi\natural}[\![{}^{l}\texttt{skip}]\!]t \stackrel{\text{def}}{=} \textsc{reset}[\![\varphi]\!](\textsc{step}(t))$
$\tau_{\mathrm{r}}^{\varphi\natural}[\![{}^{l}X := aexp]\!]t \stackrel{\text{def}}{=} \textsc{reset}[\![\varphi]\!](\textsc{b-assign}[\![X := aexp]\!]t)$
$\tau_{\mathrm{r}}^{\varphi\natural}[\![\texttt{if } {}^{l}bexp \texttt{ then } stmt_1 \texttt{ else } stmt_2 \texttt{ fi}]\!]t \stackrel{\text{def}}{=}$
$\quad \textsc{reset}[\![\varphi]\!](\textsc{filter}[\![bexp]\!](\tau_{\mathrm{r}}^{\varphi\natural}[\![stmt_1]\!]t) \curlyvee \textsc{filter}[\![\texttt{not } bexp]\!](\tau_{\mathrm{r}}^{\varphi\natural}[\![stmt_2]\!]t))$
$\tau_{\mathrm{r}}^{\varphi\natural}[\![\texttt{while } {}^{l}bexp \texttt{ do } stmt \texttt{ od}]\!]t \stackrel{\text{def}}{=} \mathrm{gfp}^{\natural}_{G(t)}\ \phi_{\mathrm{r}}^{\varphi\natural}$
$\quad G \stackrel{\text{def}}{=} \tau_{\mathrm{g}}^{\varphi\natural}[\![\texttt{while } {}^{l}bexp \texttt{ do } stmt \texttt{ od}]\!]$
$\quad \phi_{\mathrm{r}}^{\varphi\natural}(x) \stackrel{\text{def}}{=} \textsc{reset}[\![\varphi]\!](\textsc{filter}[\![bexp]\!](\tau_{\mathrm{r}}^{\varphi\natural}[\![stmt]\!]x) \curlyvee \textsc{filter}[\![\texttt{not } bexp]\!]t)$
$\tau_{\mathrm{r}}^{\varphi\natural}[\![stmt_1\ stmt_2]\!]t \stackrel{\text{def}}{=} \tau_{\mathrm{r}}^{\varphi\natural}[\![stmt_1]\!](\tau_{\mathrm{r}}^{\varphi\natural}[\![stmt_2]\!]t)$

Figure 11: Abstract recurrence semantics of instructions *stmt*.

8.3. Abstract Recurrence Semantics

We now describe the required changes to the decision trees abstract domains in order to obtain a decidable abstraction of the φ-recurrence semantics $\tau_{\mathrm{r}}^{\varphi}$ (cf. Equation 34).

We define the abstract φ-recurrence semantics $\tau_{\mathrm{r}}^{\varphi\natural} \in \mathcal{L} \to \mathcal{T}$: to each program control point $l \in \mathcal{L}$ corresponds a piecewise-defined ranking function $t \in \mathcal{T}$, and for each program instruction *stmt* a *sound* abstract recurrence semantics transformer $\tau_{\mathrm{r}}^{\varphi\natural}[\![stmt]\!] \in \mathcal{T} \to \mathcal{T}$ is defined in Figure 11. In particular, we modify the operator $\textsc{reset}[\![\varphi]\!]$ presented in the previous Section 8.2 in order to reset the value of a ranking function only when the ranking function is already defined within the pieces that satisfy a given property φ. As an example, unlike Figure 9, the decision tree in Figure 6c is unmodified by the $\textsc{reset}[\![x = 3]\!]$ operator.

The starting point of the recurrence backward analysis is now the totally undefined function at the program final control point $f[\![prog]\!]$, since the program final states cannot satisfy a recurrence property. This function is then propagated backwards towards the program initial control point $i[\![prog]\!]$. In case of loops, a first increasing iteration with widening yields their abstract guarantee semantics, which is the starting point for the decreasing iteration with a new *dual widening* operator $\bar{\triangledown}$. The dual widening $\bar{\triangledown}$ obeys:

(i) $x \sqsupseteq x \mathbin{\bar{\triangledown}} y$ and $y \sqsupseteq x \mathbin{\bar{\triangledown}} y$;

(ii) for all decreasing sequences $X_0 \sqsupseteq X_1 \sqsupseteq \cdots \sqsupseteq X_n \sqsupseteq \cdots$, the decreasing sequence $Y_0 \stackrel{\text{def}}{=} X_0$, $Y_{n+1} \stackrel{\text{def}}{=} Y_n \mathbin{\bar{\triangledown}} X_{n+1}$ stabilizes, that is, $\exists l \geq 0 : \forall j \geq l : y_j = y_l$.

Dual widenings are rather unknown and, up to our knowledge, only few practical instance has been proposed, e.g., [19, 20]. In our case, the dual widening $\bar{\triangledown}$ enforces the termination of the analysis by preventing the number of pieces of a piecewise-defined ranking function from growing indefinitely: given two piecewise-defined ranking functions $t_1 \in \mathcal{T}$ and $t_2 \in \mathcal{T}$, it enforces the piecewise-definition of the first function t_1 on the second function t_2. Then, for each piece of the ranking functions, it maintains the value of the function only if both t_1 and t_2 are defined on that piece. We propose an example of dual widening in Figure 12. In

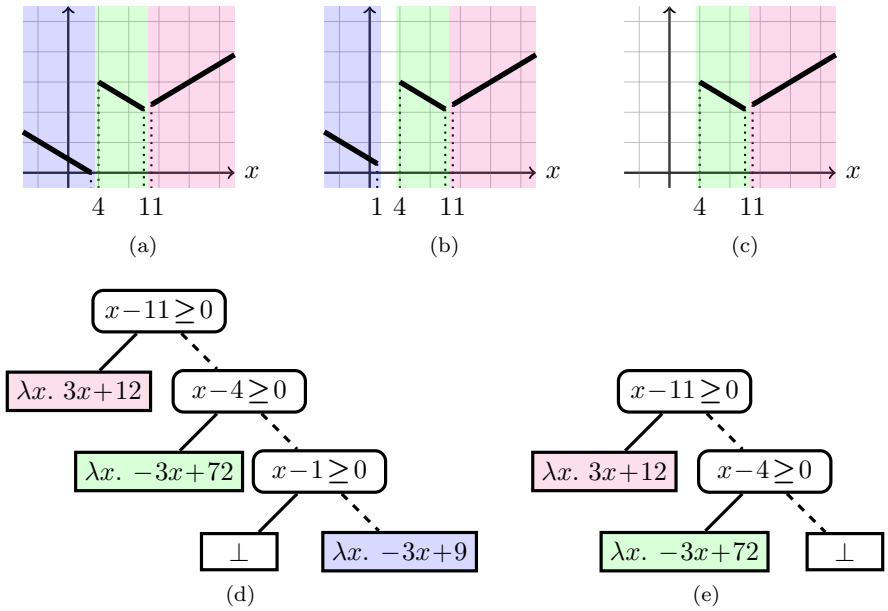


Figure 12: Dual widening between the piecewise-defined functions shown in (a) (from Figure 5b) and (b), respectively. Their decision tree representation is depicted in Figure 5a and (d). The graphical representation of the result is shown in (c) and its decision tree representation is depicted in (e).

Figure 7, $\mathrm{gfp}^{\natural}\ \phi_{\mathrm{r}}^{\varphi\natural}$ denotes the limit of the iteration sequence with dual widening:

$$y_0 \stackrel{\text{def}}{=} G(t)$$
$$y_{n+1} \stackrel{\text{def}}{=} \begin{cases} y_n & y_n \sqsubseteq^{\natural} \phi_{\mathrm{r}}^{\varphi\natural}(y_n) \wedge y_n \preccurlyeq^{\natural} \phi_{\mathrm{r}}^{\varphi\natural}(y_n) \\ y_n \mathbin{\bar{\nabla}} \phi_{\mathrm{r}}^{\varphi\natural}(y_n) & \text{otherwise} \end{cases} \tag{43}$$

The analysis computes an over-approximation of the value of the φ-recurrence semantics $\tau_{\mathrm{r}}^{\varphi}$ and an *under-approximation* of its domain of definition $\mathrm{dom}(\tau_{\mathrm{r}}^{\varphi})$. In this way, an abstraction provides *sufficient preconditions* for the recurrence property $\Box\Diamond\varphi$: if the abstraction is defined on a program state, then all the program traces branching from that state *always* reach a state with the property φ *infinitely often*, and the value of the function provides an upper bound on the number of execution steps before the next occurrence of a state with the property φ.

Example 21. In Figure 13a, we depict the decision tree inferred for proving the recurrence property $\Box\Diamond(x=3)$ at program control point 1 of the program SIMPLE of Figure 1. The graphical representation of the ranking function is shown in Figure 13b. Its domain yields the sufficient precondition $x<0$ for $\Box\Diamond(x=3)$.■

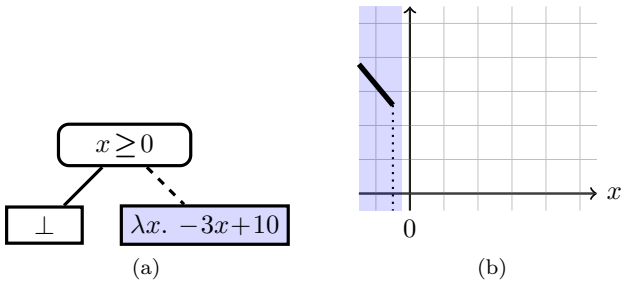


Figure 13: Decision tree representation (a) of the piecewise-defined ranking function (b) inferred for proving $\Box\Diamond(x=3)$ at program control point 1 of Figure 1.

9. Implementation

We have incorporated the static analysis methods for guarantee and recurrence temporal properties that we have presented into our prototype static analyzer FuncTion based on piecewise-defined ranking functions.

The prototype accepts (non-deterministic) programs written in a C-like syntax, without `struct` and `union` types. It provides only a limited support for arrays and pointers. The *mathematical integers* are the only basic data type, deviating from the standard semantics of C. When the guarantee or recurrence analysis methods are selected, it accepts state properties written as C-like pure expressions. The prototype is written in OCaml and, at the time of writing, the available numerical abstractions to control the pieces of the ranking functions are based on the intervals abstract domain [16], the convex polyhedra abstract domain [17], and the octagons abstract domain [18], and the available abstraction to represent the value of the ranking functions is based on affine functions. The numerical abstract domains are provided by the *APRON* library [21]. It is also possible to activate the extension to ordinal-valued ranking functions [7], and tune the precision of the analysis by adjusting the widening delay.

To improve precision, we avoid trying to compute a ranking function for the non-reachable states: FuncTion runs a forward analysis to over-approximate the reachable states using a numerical abstract domain [16, 17, 18]. Then, it runs the backward analysis to infer a ranking function, intersecting its domain at each step with the states identified by the previous analysis.

The analysis proceeds by structural induction on the program syntax, iterating loops with widening (and, for recurrence properties, both widening and dual widening) until stabilization. In case of nested loops, the analysis stabilizes the inner loop for each iteration of the outer loop, following [22].

To illustrate the effectiveness of our new static analysis methods, we consider more examples besides the program SIMPLE of Figure 1, and we present the results automatically produced by our analyzer.

Example 22. Let us consider the program COUNT-DOWN in Figure 14. Each iteration of the outer loop assigns to the variable x the value of some counter c,

```
[1]c := 1
while [2]( true ) do
  [3]x := c
  while [4](0<x) do [5]x := x−1 [6]c := c+1 od
od[7]
```

Figure 14: Program COUNT-DOWN.

```
while [1]( true ) do
  [2]x := ?
  while [3](x≠0) do
    if [4](0<x) then [5]x := x−1 else [6]x := x+1 fi
  od
od[7]
```

Figure 15: Program SINK.

which initially has value one; then, the inner loop decreases the value of x and increases the value of c until the value of x becomes less than or equal to zero.

FuncTion, parameterized by the *intervals abstract domain* [16] and using *affine functions* to represent the value of the ranking functions, is able to prove that the recurrence property $\square\diamondsuit(x=0)$ is always satisfied by the program. The piecewise-defined ranking function inferred at program control point 1 bounds the wait for the *next* occurrence of the desirable state $x=0$ by five program execution steps (i.e., executing the variable assignment $c:=1$, testing the outer loop condition, executing the assignment $x:=c$, testing the inner loop condition and executing the assignment $x:=x-1$). The analysis infers a more interesting raking function associated to program control point 4. The function bounds the wait for the next occurrence of $x=0$ by $3c+2$ execution steps when $x<0\wedge 0<c$, by 3 execution steps when $x<0\wedge c=0$ (i.e., testing the inner loop condition, testing the outer loop condition and executing the assignment $x:=c$), by 1 execution step when $x=0\wedge 0\leq c$ (i.e., testing the inner loop condition) and by $3x-1$ execution steps when $(x=1\wedge -1\leq c)\vee(2\leq x\wedge -2\leq c)$:

$$\lambda x.\ \lambda c.\ \begin{cases} 3c+2 & x<0\wedge 0<c \\ 3 & x<0\wedge c=0 \\ 1 & x=0\wedge 0\leq c \\ 3x-1 & (x=1\wedge -1\leq c)\vee(2\leq x\wedge -2\leq c) \\ \text{undefined} & \text{otherwise} \end{cases}$$

In the last case there is a precision loss due to a lack of expressiveness of the intervals abstract domain: if x is strictly positive at program control point 4, the weakest precondition ensuring infinitely many occurrences of the desirable state $x=0$ is $-x\leq c$, which is not representable by the intervals abstract domain. ■

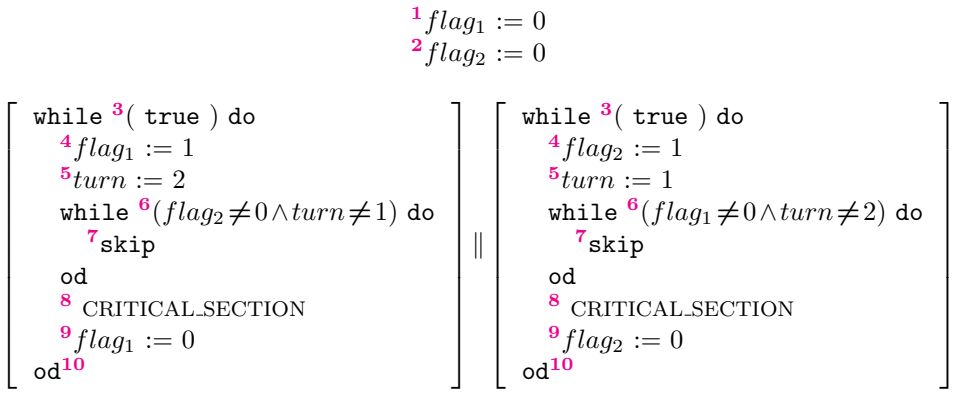


Figure 16: Program PETERSON (Peterson's Algorithm).

Example 23. Let us consider the program SINK in Figure 15. Each iteration of the outer loop resets the value of the program variable x with the non-deterministic assignment $x := ?$; then, the inner loop decreases (when x is positive) or increases (when x is negative) the value of x until it becomes equal to zero.

The recurrence property $\Box\Diamond(x = 0)$ is clearly satisfied by the program. However, because of the non-deterministic assignment $x := ?$, the number of execution steps between two occurrences of the desirable state $x = 0$ is unbounded. FUNCTION, parameterized by the *intervals abstract domain* [16] and using *ordinal-valued affine functions* [7] to represent the value of the ranking functions, is able to prove that the property is satisfied. The ranking function at program control point 1:

$$\lambda x.\ \omega{+}8$$

means that, whatever the value of x, the number of execution steps between two occurrences of $x = 0$ is unbounded but *finite*. Indeed, since ordinals are a well-ordered set, any strictly decreasing sequence starting at $\omega{+}8$ is necessarily finite. Thus, the value $\omega{+}8$ proves that $x = 0$ necessarily happens infinitely often. ■

Example 24. Let us consider the program PETERSON, Peterson's algorithm for mutual exclusion, in Figure 16. Note that *weak fairness* [9] assumptions are required in order to guarantee bounded bypass (i.e., a process cannot be bypassed by any other process in entering the critical section for more than a finite number of times). At the moment our prototype FUNCTION is not able to directly analyze concurrent programs. Thus, we have modeled the algorithm as a fair non-deterministic sequential program which interleaves execution steps from both processes while enforcing 1-bounded bypass (i.e., a process cannot be bypassed by any other process in entering the critical section for more than once). FUNCTION, parameterized by the *intervals abstract domain* [16] and using *affine functions* to represent the value of the ranking functions, is able to prove the recurrence property $\Box\Diamond(8{:}\texttt{true})$, meaning that both processes are allowed to enter their critical section infinitely often. ■

These and additional examples are available from FuncTion web interface: http://www.di.ens.fr/~urban/FuncTion.html. We refer to [8, 23] for more extensive experimental evaluations restricted to program termination.

10. Related Work

In the recent past, a large body of work has been devoted to proving liveness properties of (concurrent) programs.

A successful approach for proving liveness properties is based on a transformation from model checking of liveness properties to model checking of *safety* properties [24]. The approach looks for and exploits lasso-shaped counterexamples. A similar search for lasso-shaped counterexamples has been used to generalize the model checking algorithm IC3 to deal with liveness properties [25]. However, in general, counterexamples to liveness properties in infinite-state systems are not necessarily lasso-shaped. Our approach is not counterexample-based and is meant for proving liveness properties directly, without reduction to safety properties.

In [26], Andreas Podelski and Andrey Rybalchenko present a method for the verification of liveness properties based on transition invariants [27]. The approach, as in [28], reduces the proof of a liveness property to the proof of *fair termination* by means of a program transformation. It is at the basis of the industrial-scale tool TERMINATOR [29]. By contrast, our method is meant for proving liveness properties directly, without reduction to termination (the same argument applies to other approaches based on the size-change termination principle [30]). Moreover, our method avoids the cost of explicit checking for the well-foundedness of the transition invariants. In [31], transition invariants are computed by an off-the-shelf forward abstract interpretation, while our abstract domains are specifically dedicated to the inference of ranking functions via backward analysis.

A distinguishing aspect of our work is the use of infinite height abstract domains, equipped with (dual) widening. We are aware of only one other such work: in [32], Damien Massé proposes a method for proving arbitrary temporal properties based on abstract domains for lower closure operators. A small analyzer is presented in [33] but the approach remains mainly theoretical. We believe that our framework, albeit less general, is more straightforward and of practical use.

An emerging trend focuses on proving *existential* temporal properties (e.g., proving that there exists a particular execution trace). The most recent approaches [34, 35] are based on counterexample-guided abstraction refinement [36]. Our work is designed for proving universal temporal properties (i.e., valid for all program execution traces). We leave proving existential temporal properties, as well as proving more expressive temporal properties such as CTL and CTL* properties [37, 38], as part of our future work.

Finally, to our knowledge, the inference of sufficient preconditions for guarantee and recurrence properties (as opposed to proving a temporal property unconditionally), and the ability to provide upper bounds on the waiting time before a program reaches a desirable state, are unique to our work. In particular, the ability to infer preconditions is key to enable *modular* analyses, which allow

reasoning on a portion of the code at a time without any knowledge about its context in the complete program. We are aware of only few works that have addressed the problem of finding sufficient preconditions for program termination. In [39], preconditions are acquired in order to strengthen a termination argument, while our preconditions are inherently obtained from the inferred ranking functions as the set of program states for which the ranking function is defined. Thus, our preconditions are derived by under-approximation of the set of terminating states as opposed to the approaches presented in [40, 41] where the preconditions are derived by (complementing an) over-approximation of the non-terminating states.

11. Conclusion and Future Work

In this paper, we have presented an abstract interpretation framework for proving *guarantee* and *recurrence* temporal properties of programs. We have systematically derived by abstract interpretation new sound static analysis methods to effectively infer *sufficient preconditions* for these temporal properties, and to provide *upper bounds* on the wait before a program reaches a desirable state.

It remains for future work to explicitly express and handle fairness properties. Another natural future direction is analyzing concurrent programs directly, without resorting to their sequential encoding. We also plan to extend the present framework to the full hierarchy of temporal properties presented in [2] (to the class of persistence properties, in particular) and more generally to *arbitrary* (universal and existential) liveness properties.

Acknowledgements. The research leading to these results has received funding from the French Agence Nationale de la Recherche (project ANR-11-INSE-014).

[1] L. Lamport, Proving the Correctness of Multiprocess Programs, IEEE Transactions on Software Engineering 3 (2) (1977) 125–143.

[2] Z. Manna, A. Pnueli, A Hierarchy of Temporal Properties, in: PODC, 1990, pp. 377–410.

[3] P. Cousot, R. Cousot, Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, in: POPL, 1977, pp. 238–252.

[4] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, X. Rival, Static Analysis and Verification of Aerospace Software by Abstract Interpretation, in: AIAA, 2010, pp. 1–38.

[5] P. Cousot, R. Cousot, An Abstract Interpretation Framework for Termination, in: POPL, 2012, pp. 245–258.

[6] C. Urban, The Abstract Domain of Segmented Ranking Functions, in: SAS, 2013, pp. 43–62.

[7] C. Urban, A. Miné, An Abstract Domain to Infer Ordinal-Valued Ranking Functions, in: ESOP, 2014, pp. 412–431.

[8] C. Urban, A. Miné, A Decision Tree Abstract Domain for Proving Conditional Termination, in: SAS, 2014, pp. 302–318.

[9] N. Francez, Fairness, Springer, 1986.

[10] C. Urban, A. Miné, Proving Guarantee and Recurrence Temporal Properties by Abstract Interpretation, in: VMCAI, 2015, pp. 190–208.

[11] P. Cousot, Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation, Electronic Notes in Theoretical Computer Science 6 (1997) 77–102.

[12] A. Turing, Checking a Large Routine, in: Report of a Conference on High Speed Automatic Calculating Machines, 1949, pp. 67–69.

[13] R. W. Floyd, Assigning Meanings to Programs, Proceedings of Symposium on Applied Mathematics 19 (1967) 19–32.

[14] P. Cousot, R. Cousot, Higher Order Abstract Interpretation (and Application to Comportment Analysis Generalizing Strictness, Termination, Projection, and PER Analysis, in: ICCL, 1994, pp. 95–112.

[15] D. Park, Fixpoint Induction and Proofs of Program Properties, Machine Intelligence 5 (1969) 59–78.

[16] P. Cousot, R. Cousot, Static Determination of Dynamic Properties of Programs, in: Symposium on Programming, 1976, pp. 106–130.

[17] P. Cousot, N. Halbwachs, Automatic Discovery of Linear Restraints Among Variables of a Program, in: POPL, 1978, pp. 84–96.

[18] A. Miné, The Octagon Abstract Domain, Higher-Order and Symbolic Computation 19 (1) (2006) 31–100.

[19] A. Chakarov, S. Sankaranarayanan, Expectation Invariants for Probabilistic Program Loops as Fixed Points, in: SAS, 2014, pp. 85–100.

[20] A. Miné, Inferring Sufficient Conditions with Backward Polyhedral Under-Approximations, in: NSAD, Vol. 287 of Electronic Notes in Theoretical Computer Science, 2012, pp. 89–100.

[21] B. Jeannet, A. Miné, Apron: A Library of Numerical Abstract Domains for Static Analysis, in: CAV, 2009, pp. 661–667.

[22] F. Bourdoncle, Efficient Chaotic Iteration Strategies with Widenings, in: FMPA, 1993, pp. 128–141.

[23] V. D'Silva, C. Urban, Conflict-Driven Conditional Termination, in: CAV (II), 2015, pp. 271–286.

[24] A. Biere, C. Artho, V. Schuppan, Liveness Checking as Safety Checking, Electronic Notes in Theoretical Computer Science 66 (2) (2002) 160–177.

[25] A. R. Bradley, F. Somenzi, Z. Hassan, Y. Zhang, An Incremental Approach to Model Checking Progress Properties, in: FMCAD, 2011, pp. 144–153.

[26] A. Podelski, A. Rybalchenko, Transition Predicate Abstraction and Fair Termination, in: POPL, 2005, pp. 132–144.

[27] A. Podelski, A. Rybalchenko, Transition Invariants, in: LICS, 2004, pp. 32–41.

[28] M. Y. Vardi, Verification of Concurrent Programs: The Automata-Theoretic Framework, Annals of Pure and Applied Logic 51 (1-2) (1991) 79–98.

[29] B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, M. Y. Vardi, Proving that Programs Eventually do Something Good, in: POPL, 2007, pp. 265–276.

[30] C. S. Lee, N. D. Jones, A. M. Ben-Amram, The Size-Change Principle for Program Termination, in: POPL, 2001, pp. 81–92.

[31] J. Berdine, A. Chawdhary, B. Cook, D. Distefano, P. W. O'Hearn, Variance Analyses from Invariance Analyses, in: POPL, 2007, pp. 211–224.

[32] D. Massé, Property Checking Driven Abstract Interpretation-Based Static Analysis, in: VMCAI, 2003, pp. 56–69.

[33] D. Massé, Abstract Domains for Property Checking Driven Analysis of Temporal Properties, in: AMAST, 2004, pp. 349–363.

[34] T. A. Beyene, C. Popeea, A. Rybalchenko, Solving Existentially Quantified Horn Clauses, in: CAV, 2013, pp. 869–882.

[35] B. Cook, E. Koskinen, Reasoning About Nondeterminism in Programs, in: PLDI, 2013, pp. 219–230.

[36] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-Guided Abstraction Refinement for Symbolic Model Checking, Journal of the ACM 50 (5) (2003) 752–794.

[37] B. Cook, H. Khlaaf, N. Piterman, Faster Temporal Reasoning for Infinite-State Programs, in: FMCAD, 2014, pp. 75–82.

[38] B. Cook, H. Khlaaf, N. Piterman, On Automation of CTL* Verification for Infinite-State Systems, in: CAV (I), 2015, pp. 13–29.

[39] B. Cook, S. Gulwani, T. Lev-Ami, A. Rybalchenko, M. Sagiv, Proving Conditional Termination, in: CAV, 2008, pp. 328–340.

[40] P. Ganty, S. Genaim, Proving Termination Starting from the End, in: CAV, 2013, pp. 397–412.

[41] D. Massé, Policy Iteration-based Conditional Termination and Ranking Functions, in: VMCAI, 2014, pp. 453–471.

$$
\begin{array}{lcl}
[\![X]\!]\rho & \stackrel{\text{def}}{=} & \{\rho(X)\} \\
[\![[a,b]]\!]\rho & \stackrel{\text{def}}{=} & \{x \mid a \leq x \leq b\} \\
[\![-aexp]\!]\rho & \stackrel{\text{def}}{=} & \{-x \mid x \in [\![aexp]\!]\rho\} \\
[\![aexp_1 + aexp_2]\!]\rho & \stackrel{\text{def}}{=} & \{x+y \mid x \in [\![aexp_1]\!], y \in [\![aexp_2]\!]\rho\} \\
[\![aexp_1 - aexp_2]\!]\rho & \stackrel{\text{def}}{=} & \{x-y \mid x \in [\![aexp_1]\!], y \in [\![aexp_2]\!]\rho\} \\
[\![aexp_1 * aexp_2]\!]\rho & \stackrel{\text{def}}{=} & \{x*y \mid x \in [\![aexp_1]\!], y \in [\![aexp_2]\!]\rho\} \\
[\![aexp_1 \,/\, aexp_2]\!]\rho & \stackrel{\text{def}}{=} & \{trnk(x/y) \mid x \in [\![aexp_1]\!], y \in [\![aexp_2]\!]\rho, y \neq 0\}
\end{array}
$$

where $trnk\colon \mathbb{R} \to \mathbb{Z}$ is defined as follows:

$$
trnk(x) \stackrel{\text{def}}{=} \begin{cases} \max\{y \in \mathbb{Z} \mid y \leq x\} & x \geq 0 \\ \min\{y \in \mathbb{Z} \mid y \geq x\} & x < 0 \end{cases}
$$

Figure A.17: Semantics of arithmetic expressions *aexp*.

$$
\begin{array}{lcl}
[\![\texttt{?}]\!]\rho & \stackrel{\text{def}}{=} & \{\text{true},\text{false}\} \\
[\![\texttt{not}\, bexp]\!]\rho & \stackrel{\text{def}}{=} & \{\neg x \mid x \in [\![bexp]\!]\rho\} \\
[\![bexp_1 \,\texttt{and}\, bexp_2]\!]\rho & \stackrel{\text{def}}{=} & \{x \wedge y \mid x \in [\![bexp_1]\!], y \in [\![bexp_2]\!]\rho\} \\
[\![bexp_1 \,\texttt{or}\, bexp_2]\!]\rho & \stackrel{\text{def}}{=} & \{x \vee y \mid x \in [\![bexp_1]\!], y \in [\![bexp_2]\!]\rho\} \\
[\![aexp_1 < aexp_2]\!]\rho & \stackrel{\text{def}}{=} & \{x < y \mid x \in [\![aexp_1]\!], y \in [\![aexp_2]\!]\rho\} \\
[\![aexp_1 \leq aexp_2]\!]\rho & \stackrel{\text{def}}{=} & \{x \leq y \mid x \in [\![aexp_1]\!], y \in [\![aexp_2]\!]\rho\} \\
[\![aexp_1 = aexp_2]\!]\rho & \stackrel{\text{def}}{=} & \{x = y \mid x \in [\![aexp_1]\!], y \in [\![aexp_2]\!]\rho\} \\
[\![aexp_1 \neq aexp_2]\!]\rho & \stackrel{\text{def}}{=} & \{x \neq y \mid x \in [\![aexp_1]\!], y \in [\![aexp_2]\!]\rho\}
\end{array}
$$

Figure A.18: Semantics of boolean expressions *bexp*.

Appendix A. Language Semantics

The semantics $[\![aexp]\!]\colon \mathcal{E} \to \mathcal{P}(\mathbb{Z})$ of an arithmetic expression *aexp* is defined in Figure A.17. Similarly, the semantics $[\![bexp]\!]\colon \mathcal{E} \to \mathcal{P}(\{\text{true},\text{false}\})$ of a boolean expression *bexp* is presented in Figure A.18. The initial control point $i[\![stmt]\!] \in \mathcal{L}$ (resp. $i[\![prog]\!] \in \mathcal{L}$) of an instruction *stmt* (resp. a program *prog*), and the final control point $f[\![stmt]\!] \in \mathcal{L}$ (resp. $f[\![prog]\!] \in \mathcal{L}$) of an instruction *stmt* (resp. a program *prog*) are formally defined in Figure A.19 and Figure A.20, respectively.

Appendix B. Proofs

Theorem 2. *A program satisfies a guarantee property $\Diamond\varphi$ for all traces starting from a given set of initial states $\mathcal{I}$ if and only if $\mathcal{I} \subseteq \mathrm{dom}(\tau_g^\varphi)$.*

Proof. The proof follows by Park's Fixpoint Induction Principle [15]. More specifically, we have $\mathcal{I} \subseteq \mathrm{dom}(\tau_{\mathrm{g}}^\varphi)$ if and only if $\exists v\colon \Sigma \rightharpoonup \mathbb{O}\colon \tau_{\mathrm{g}}^\varphi \sqsubseteq v \wedge \mathcal{I} \subseteq \mathrm{dom}(v)$.

$$
\begin{array}{lcll}
\mathit{stmt} & ::= & {}^{l}\texttt{skip} & i[\![{}^{l}\texttt{skip}]\!] \stackrel{\text{def}}{=} l \\
& | & {}^{l}X := \mathit{aexp} & i[\![{}^{l}X := \mathit{aexp}]\!] \stackrel{\text{def}}{=} l \\
& | & \texttt{if } {}^{l}\mathit{bexp} \texttt{ then } \mathit{stmt}_1 \texttt{ else } \mathit{stmt}_2 \texttt{ fi} & \\
& & \qquad i[\![\texttt{if } {}^{l}\mathit{bexp} \texttt{ then } \mathit{stmt}_1 \texttt{ else } \mathit{stmt}_2 \texttt{ fi}]\!] \stackrel{\text{def}}{=} l \\
& | & \texttt{while } {}^{l}\mathit{bexp} \texttt{ do } \mathit{stmt}_1 \texttt{ od} & \\
& & \qquad i[\![\texttt{while } {}^{l}\mathit{bexp} \texttt{ do } \mathit{stmt}_1 \texttt{ od}]\!] \stackrel{\text{def}}{=} l \\
& | & \mathit{stmt}_1 \ \mathit{stmt}_2 & i[\![\mathit{stmt}_1 \ \mathit{stmt}_2]\!] \stackrel{\text{def}}{=} i[\![\mathit{stmt}_1]\!] \\
\mathit{prog} & ::= & \mathit{stmt}\ {}^{l} & i[\![\mathit{stmt}\ {}^{l}]\!] \stackrel{\text{def}}{=} i[\![\mathit{stmt}]\!]
\end{array}
$$

Figure A.19: Initial control point of instructions *stmt* and programs *prog*.

$$
\begin{array}{lcll}
\mathit{stmt} & ::= & {}^{l}\texttt{skip} & f[\![{}^{l}\texttt{skip}]\!] \stackrel{\text{def}}{=} f[\![\mathit{stmt}]\!] \\
& | & {}^{l}X := \mathit{aexp} & f[\![{}^{l}X := \mathit{aexp}]\!] \stackrel{\text{def}}{=} f[\![\mathit{stmt}]\!] \\
& | & \texttt{if } {}^{l}\mathit{bexp} \texttt{ then } \mathit{stmt}_1 \texttt{ else } \mathit{stmt}_2 \texttt{ fi} & \\
& & \qquad f[\![\texttt{if } {}^{l}\mathit{bexp} \texttt{ then } \mathit{stmt}_1 \texttt{ else } \mathit{stmt}_2 \texttt{ fi}]\!] \stackrel{\text{def}}{=} f[\![\mathit{stmt}]\!] \\
& & \qquad\qquad f[\![\mathit{stmt}_1]\!] \stackrel{\text{def}}{=} f[\![\mathit{stmt}]\!] \\
& & \qquad\qquad f[\![\mathit{stmt}_2]\!] \stackrel{\text{def}}{=} f[\![\mathit{stmt}]\!] \\
& | & \texttt{while } {}^{l}\mathit{bexp} \texttt{ do } \mathit{stmt}_1 \texttt{ od} & \\
& & \qquad f[\![\texttt{while } {}^{l}\mathit{bexp} \texttt{ do } \mathit{stmt}_1 \texttt{ od}]\!] \stackrel{\text{def}}{=} f[\![\mathit{stmt}]\!] \\
& & \qquad\qquad f[\![\mathit{stmt}_1]\!] \stackrel{\text{def}}{=} l \\
& | & \mathit{stmt}_1 \ \mathit{stmt}_2 & f[\![\mathit{stmt}_1 \ \mathit{stmt}_2]\!] \stackrel{\text{def}}{=} f[\![\mathit{stmt}]\!] \\
& & \qquad\qquad f[\![\mathit{stmt}_1]\!] \stackrel{\text{def}}{=} i[\![\mathit{stmt}_2]\!] \\
& & \qquad\qquad f[\![\mathit{stmt}_2]\!] \stackrel{\text{def}}{=} f[\![\mathit{stmt}]\!] \\
\mathit{prog} & ::= & \mathit{stmt}\ {}^{l} & f[\![\mathit{stmt}\ {}^{l}]\!] \stackrel{\text{def}}{=} l
\end{array}
$$

Figure A.20: Final control point of instructions *stmt* and programs *prog*.

Then we have:

$$
\begin{aligned}
\tau_{\mathrm{g}}^{\varphi} \sqsubseteq v \ &\Leftrightarrow\ \mathrm{lfp}_{\dot{\emptyset}}^{\sqsubseteq}\ \phi_{\mathrm{g}}[\varphi] \sqsubseteq v && \text{(from Equation 22 and Equation 21)} \\
&\Leftrightarrow\ \exists v' \colon \Sigma \rightharpoonup \mathbb{O} \colon \dot{\emptyset} \sqsubseteq v' \wedge \phi_{\mathrm{g}}[\varphi](v') \sqsubseteq v' \wedge v' \sqsubseteq v \\
& && \text{(from Park's Fixpoint Induction Principle [15])} \\
&\Leftrightarrow\ \phi_{\mathrm{g}}[\varphi](v) \sqsubseteq v && \text{(by definition of } \sqsubseteq \text{ and choosing } v' = v) \\
&\Leftrightarrow\ \mathrm{dom}(\phi_{\mathrm{g}}[\varphi](v)) \subseteq \mathrm{dom}(v) \wedge \forall s \in \mathrm{dom}(\phi_{\mathrm{g}}[\varphi](v)) \colon \phi_{\mathrm{g}}[\varphi](v)s \leq v(s) \\
& && \text{(by definition of } \sqsubseteq) \\
&\Leftrightarrow\ \forall s \in \mathrm{dom}(v) \colon (\exists s' \in \mathrm{dom}(v) \colon \langle s, s' \rangle \in \tau) \Rightarrow \\
& \quad \forall s' \in \Sigma \colon \langle s, s' \rangle \in \tau \Rightarrow s' \in \mathrm{dom}(v) \wedge v(s') < v(s) \\
& && \text{(by definition of } \phi_{\mathrm{g}}[\varphi], \text{ cf. Equation 21)}
\end{aligned}
$$

Now, from Definition 6, $v : \Sigma \rightharpoonup \mathbb{O}$ is a ranking function. Thus, choosing $\mathcal{I} \subseteq \mathrm{dom}(v)$, concludes the proof. □

Theorem 3. *A program satisfies a recurrence property* $\square\Diamond\varphi$ *for all traces starting*

from a given set of states $\mathcal{I}$ if and only if $\mathcal{I} \subseteq \mathrm{dom}(\tau_r^\varphi)$.

Proof. The proof follows from the dual of Park's Fixpoint Induction Principle [15] and from Theorem 2. More specifically, we have $\mathcal{I} \subseteq \mathrm{dom}(\tau_{\mathrm{r}}^\varphi)$ if and only if $\exists v \colon \Sigma \rightharpoonup \mathbb{O} : v \sqsubseteq \tau_{\mathrm{r}}^\varphi \wedge \mathcal{I} \subseteq \mathrm{dom}(v)$. Then we have:

$$
\begin{aligned}
v \sqsubseteq \tau_{\mathrm{r}}^\varphi \;&\Leftrightarrow\; v \sqsubseteq \mathrm{gfp}^{\sqsubseteq}_{\tau_{\mathrm{g}}[\varphi]}\, \phi_{\mathrm{r}}[\varphi] && \text{(from Equation 34 and Equation 33)} \\
&\Leftrightarrow\; \exists v' \colon \Sigma \rightharpoonup \mathbb{O} : v' \sqsubseteq \tau_{\mathrm{g}}[\varphi] \wedge v' \sqsubseteq \phi_{\mathrm{r}}[\varphi](v') \wedge v \sqsubseteq v' \\
&&& \text{(from the dual of Park's Fixpoint Induction Principle [15])} \\
&\Leftrightarrow\; v \sqsubseteq \phi_{\mathrm{r}}[\varphi](v) \sqsubseteq \tau_{\mathrm{g}}[\varphi] && \text{(by definition of } \sqsubseteq \text{ and choosing } v' = v) \\
&\Leftrightarrow\; \mathrm{dom}(v) \subseteq \mathrm{dom}(\phi_{\mathrm{r}}[\varphi](v)) \subseteq \mathrm{dom}(\tau_{\mathrm{g}}[\varphi]) \\
&\quad \wedge \forall s \in \mathrm{dom}(v) : v(s) \leq \phi_{\mathrm{r}}[\varphi](v) s \leq \tau_{\mathrm{g}}[\varphi](s) && \text{(by definition of } \sqsubseteq) \\
&\Leftrightarrow\; \forall s \in \mathrm{dom}(v) : (\exists s' \in \mathrm{dom}(v) : \langle s, s' \rangle \in \tau) \Rightarrow \\
&\quad \forall s' \in \Sigma : \langle s, s' \rangle \in \tau \Rightarrow s' \in \mathrm{dom}(v) \wedge v(s') \leq \tau_{\mathrm{g}}[\varphi](s') < v(s) \leq \tau_{\mathrm{g}}[\varphi](s) \\
&&& \text{(by definition of } \phi_{\mathrm{r}}[\varphi], \text{ cf. Equation 33)}
\end{aligned}
$$

Now, from Theorem 2 and Definition 6, $v \colon \Sigma \rightharpoonup \mathbb{O}$ is a ranking function. Thus, choosing $\mathcal{I} \subseteq \mathrm{dom}(v)$, concludes the proof. □