The Abstract Domain of Segmented Ranking Functions

Caterina Urban

The Abstract Domain of Segmented Ranking Functions

Caterina Urban

École Normale Supérieure - CNRS - INRIA, Paris, France
urban@di.ens.fr

Abstract. We present a parameterized abstract domain for proving program termination by abstract interpretation. The domain automatically synthesizes piecewise-defined ranking functions and infers sufficient conditions for program termination. The analysis uses over-approximations but we prove its soundness, meaning that all program executions respecting these sufficient conditions are indeed terminating.
The abstract domain is parameterized by a numerical abstract domain for environments and a numerical abstract domain for functions. This parameterization allows to easily tune the trade-off between precision and cost of the analysis. We describe an instantiation of this generic domain with intervals and affine functions. We define all abstract operators, including widening to ensure convergence.
To illustrate the potential of the proposed framework, we have implemented a research prototype static analyzer, for a small imperative language, that yielded interesting preliminary results.

1 Introduction

Static analysis has made great progress since the introduction of Abstract Interpretation [10,12]. Most results in this area are concerned with the verification of safety properties. The verification of liveness properties (and, in particular, termination) has received considerable attention recently.

The traditional method for proving program termination is based on the synthesis of ranking functions, which map program states to elements of a well-founded domain. Termination is guaranteed if a ranking function that decreases during computation is found. In [14], Patrick Cousot and Radhia Cousot proposed a unifying point of view on the existing approaches to termination, and introduced the idea of the computation of a ranking function by abstract interpretation. We build our work on their proposed general framework, and we design and implement a suitable parameterized abstract domain for proving termination of imperative programs by abstract interpretation.

The domain automatically synthesizes piecewise-defined ranking functions through backward invariance analysis. The analysis does not rely on assumptions about the structure of the analyzed program: for example, is not limited to simple loops, as in [22]. The ranking functions can be used to give upper bounds on the computational complexity of the program in terms of execution steps. Moreover,

the domain infers sufficient conditions for program termination. The analysis uses over-approximations but we prove its soundness, meaning that all program executions respecting these sufficient conditions are indeed terminating, while a program execution that does not respect these conditions might not terminate.

We employ segmentations to handle disjunctions arising in static program analysis, as proposed in [15] for array content analysis. The analysis automatically partitions the space of values for the program variables by means of abstract environments. A segment is a pair of an abstract environment and an abstract function. During the analysis (similarly to other partitioning approaches in static analysis [19,24]), segments are split by tests, modified by assignment and joined when merging control flows. Widening limits the number of segments of a ranking function to a maximum given as a parameter of the analysis.

The segmented ranking functions abstract domain is parameterized by the choice of the abstract environments (e.g. intervals, as in Section 3.1) and the choice of the abstract functions (e.g. affine functions, as in Section 3.2). This parameterization allows a wide range of instantiations of the domain making it possible to easily tune the trade-off between analysis precision and cost.

Motivating Example. To illustrate the potential of segmentations, let us consider the following program annotated with numbered labels to denote control points:

$$\begin{array}{l} \mathsf{while}\ {}^{1}(x \geq 0)\ \mathsf{do} \\ \quad {}^{2}x := -2x + 10 \\ \mathsf{od}^{3} \end{array}$$

The program terminates if we consider variables with integer values (if we admit non-integer values, for $x = \frac{10}{3}$ the program is not terminating). However, it does not have a linear ranking function. As a result, well-known methods to synthesize ranking functions like [22,5], would not be capable to guarantee its termination.

Figure 1 illustrates the details of our backward invariance analysis. We will map each program control point to a function $f \in \mathbb{Z} \mapsto \mathbb{N}$ of the (integer-valued) variable x, representing an upper bound on the number of execution steps before termination. We denote by $2[x \geq 0]$ the function obtained from the test $x \geq 0$ applied to the function at program point 2. Similarly, $3[x < 0]$ denotes the function obtained from the test $x < 0$ applied to the function at program point 3.

The analysis is performed backwards starting with the totally undefined function $\bot$ at each program point. The first iteration begins from the total function $f(x) = 0$ at program point 3. The test $x < 0$ enforces loop exit: it splits the domain of the function and enforces termination in 1 step. At program point 1, the function $3[x < 0]$ is unmodified by the join with the yet totally undefined function $2[x \geq 0]$. At program point 2, the assignment $x := -2x + 10$ propagates the function increasing its value to 2. Then, the test $x \geq 0$, since it does not need to split further the function domain, just propagates the function increasing again its value to 3. Finally, a second iteration starts joining once more the functions $3[x < 0]$ and $2[x \geq 0]$ at program point 1.

		1st iteration	2nd iteration	...	5th/6th iteration
3	$\bot$	$f(x) = 0$	$f(x) = 0$	...	$f(x) = 0$
$3[x < 0]$	$\bot$	$f(x) = \begin{cases} 1 & x < 0 \\ \bot & x \geq 0 \end{cases}$	$f(x) = \begin{cases} 1 & x < 0 \\ \bot & x \geq 0 \end{cases}$	...	$f(x) = \begin{cases} 1 & x < 0 \\ \bot & x \geq 0 \end{cases}$
1	$\bot$	$f(x) = \begin{cases} 1 & x < 0 \\ \bot & x \geq 0 \end{cases}$	$f(x) = \begin{cases} 1 & x < 0 \\ \bot & 0 \leq x \leq 5 \\ 3 & x > 5 \end{cases}$	...	$f(x) = \begin{cases} 1 & x < 0 \\ 5 & 0 \leq x \leq 2 \\ 9 & x = 3 \\ 7 & 4 \leq x \leq 5 \\ 3 & x > 5 \end{cases}$
2	$\bot$	$f(x) = \begin{cases} \bot & x \leq 5 \\ 2 & x > 5 \end{cases}$	$f(x) = \begin{cases} 4 & x \leq 2 \\ \bot & 3 \leq x \leq 5 \\ 2 & x > 5 \end{cases}$	...	$f(x) = \begin{cases} 4 & x \leq 2 \\ 8 & x = 3 \\ 6 & 4 \leq x \leq 5 \\ 2 & x > 5 \end{cases}$
$2[x \geq 0]$	$\bot$	$f(x) = \begin{cases} \bot & x \leq 5 \\ 3 & x > 5 \end{cases}$	$f(x) = \begin{cases} \bot & x < 0 \\ 5 & 0 \leq x \leq 2 \\ \bot & 3 \leq x \leq 5 \\ 3 & x > 5 \end{cases}$	...	$f(x) = \begin{cases} \bot & x < 0 \\ 5 & 0 \leq x \leq 2 \\ 9 & x = 3 \\ 7 & 4 \leq x \leq 5 \\ 3 & x > 5 \end{cases}$

Fig. 1: Motivating Example Analysis. The analysis starts from $f(x) = 0$ at program point 3. At program point 1, the functions $3[x < 0]$ (obtained from the test $x < 0$) and $2[x \geq 0]$ (obtained from the test $x \geq 0$) are joined.

In this particular case, there is no need for convergence acceleration and the analysis is rather precise: at the sixth iteration, a fix-point is reached providing the following ranking function $f \in \mathbb{Z} \mapsto \mathbb{N}$ as loop invariant at program point 1:

$$f(x) = \begin{cases} 1 & x < 0 \\ 5 & 0 \leq x \leq 2 \\ 9 & x = 3 \\ 7 & 4 \leq x \leq 5 \\ 3 & x > 5 \end{cases}$$

Unlike [22,5], our method is not impaired from the fact that the program does not have a linear ranking function.

Our Contribution. In summary, this paper proposes a new abstract domain for proving termination of imperative programs. We introduce the family of parameterized abstract domains of segmented ranking functions (Section 3). We also describe the design (Section 3.3) and implementation (Section 4) of a particular instance of these generic domains based on affine functions.

Outline of the Paper. Section 2 introduces the syntax and concrete semantics of our language. In Section 3 we define the segmented ranking functions abstract domain. We describe the implementation of our prototype static analyzer, in Section 4. Finally, Section 5 discusses related work and Section 6 concludes and envisions future work.

2 Concrete Termination Semantics

In the following, we briefly recall some results presented in a language independent way in [14]. Then, we tailor these results for a small imperative language.

2.1 Termination Semantics

We consider a programming language with non-deterministic programs. We describe the small-step operational semantics of a program by means of a transition system $\langle \Sigma, \tau \rangle$. Σ is the set of all program states, and $\tau \subseteq \Sigma \times \Sigma$ is the transition relation: a binary relation describing the transitions between a state and its possible successors during program execution. Let β_τ denote the set of final states: $\beta_\tau \triangleq \{s \in \Sigma \mid \forall s' \in \Sigma : \langle s, s' \rangle \notin \tau\}$. The program trace semantics generated by a transition system $\langle \Sigma, \tau \rangle$ is the set of all infinite traces over the states in Σ and all finite traces that end with a final state in β_τ.

The traditional method for proving program termination is based on ranking functions, mapping program states to elements of a well-founded domain (e.g., ordinals in $\mathbb{O}$). Termination is guaranteed if a ranking function that decreases during computation is found.

In [14], Patrick Cousot and Radhia Cousot prove the existence of a most precise ranking function that can be expressed in fix-point form by abstract interpretation of the program trace semantics. This function[1] $v_\tau \in \Sigma \rightharpoonup \mathbb{O}$ associates to each program state definitely leading to a final state in β_τ (i.e. a program state such that all traces to which it belongs end up at a final state in β_τ), an ordinal in $\mathbb{O}$ representing an upper bound on the number of remaining program execution steps to termination. Otherwise stated, v_τ is a partial function which domain $\mathsf{dom}(v_\tau)$ is the set of states leading to program termination: any trace starting in a state $s \in \mathsf{dom}(v_\tau)$ must terminate in at most $v_\tau(s)$ execution steps, while a trace starting in a state $s \notin \mathsf{dom}(v_\tau)$ might not terminate.

Let us define a computational partial order $\preccurlyeq$:

$$v_1 \preccurlyeq v_2 \triangleq \mathsf{dom}(v_1) \subseteq \mathsf{dom}(v_2) \wedge \forall x \in \mathsf{dom}(v_1) : v_1(x) \leq v_2(x).$$

Its related join operator is:

$$v_1 \curlyvee v_2 \triangleq \lambda\rho. \begin{cases} v_1(\rho) & \text{if } \rho \in \mathsf{dom}(v_1) \setminus \mathsf{dom}(v_2) \\ \mathsf{sup}\{v_1(\rho), v_2(\rho)\} & \text{if } \rho \in \mathsf{dom}(v_1) \cap \mathsf{dom}(v_2) \\ v_2(\rho) & \text{if } \rho \in \mathsf{dom}(v_2) \setminus \mathsf{dom}(v_1) \end{cases}$$

[1] $A \rightharpoonup B$ is the set of partial maps from a set A to a set B.

Then, the ranking function v_τ is computed by fix-point iteration[2] starting from the totally undefined function $\dot{\emptyset}$:

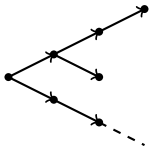
$$v_\tau \triangleq \mathsf{lfp}^{\preccurlyeq}_{\dot{\emptyset}}\ \phi_\tau$$

$$\phi_\tau(v) \triangleq \lambda s. \begin{cases} 0 & \text{if } s \in \beta_\tau \\ \mathsf{sup}\{v(s')+1 \mid \langle s, s'\rangle \in \tau\} & \text{if } s \in \widetilde{\mathsf{pre}}(\mathsf{dom}(v)) \end{cases}$$

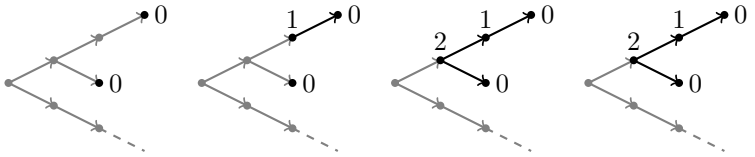
The idea is to extract the well-founded part of the transition relation τ: starting from the states in β_τ and, through a backward computation based on the inverse of the transition relation τ, mapping all the states definitely leading to a final state to their ordinal rank. In case of a non-deterministic transition system $\langle \Sigma, \tau\rangle$, using $\widetilde{\mathsf{pre}}$ ensures that we take into account all the possibly infinite choices made at each execution step, eliminating all traces potentially branching (through local non-determinism) to non-termination.

The next example is taken from [14].

Example 1. Let us consider the following trace semantics:



The fix-point iterates for the corresponding ranking function are:



□

Note that the computational order $\preccurlyeq$ does not coincide (see [12,13] for further discussion) with the approximation order $\sqsubseteq$. The order $\sqsubseteq$ is defined as follows:

$$v_1 \sqsubseteq v_2 \triangleq \mathsf{dom}(v_1) \supseteq \mathsf{dom}(v_2) \wedge \forall x \in \mathsf{dom}(v_2) : v_1(x) \leq v_2(x).$$

Its corresponding join operator is:

$$v_1 \sqcup v_2 \triangleq \lambda \rho \in \mathsf{dom}(v_1) \cap \mathsf{dom}(v_2).\ \mathsf{sup}\{v_1(\rho), v_2(\rho)\}.$$

The partial orders coincide only when the functions have the same domain:

Lemma 1. $\mathsf{dom}(v_1) = \mathsf{dom}(v_2) \Rightarrow (v_1 \sqsubseteq v_2 \Leftrightarrow v_1 \preccurlyeq v_2)$.

[2] $\widetilde{\mathsf{pre}}(X) \triangleq \{s \in \Sigma \mid \forall s' \in \Sigma : \langle s, s'\rangle \in \tau \Rightarrow s' \in X\}$.

$$
\begin{array}{lll}
X \in \mathcal{X},\ n, n_1, n_2 \in \mathbb{I} & & \\
A ::= & X \mid n \mid [n_1, n_2] \mid ? \mid -A \mid A_1 \diamond A_2 & \diamond \in \{+, -, *, /\} \\
B ::= & \mathsf{true} \mid \mathsf{false} \mid ? \mid !B \mid B_1 \vee B_2 \mid B_1 \wedge B_2 \mid A_1 \bowtie A_2 & \bowtie \in \{<, \leq, =, \neq, >, \geq\} \\
S ::= & X := A \mid \mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi} \mid \mathsf{while}\ B\ \mathsf{do}\ S\ \mathsf{od} \mid S_1; S_2 &
\end{array}
$$

Fig. 2: Syntax

The termination semantics v_τ is sound and complete to prove termination of a program P with initial states in I:

Theorem 1. *A program P, with trace semantics generated by a transition system $\langle \Sigma, \tau \rangle$, terminates for all traces starting from initial states in $I \in \wp(\Sigma)$ if and only if there exists $v \in \Sigma \rightharpoonup \mathbb{O}$ such that $v_\tau \sqsubseteq v \wedge I \subseteq \mathsf{dom}(v)$.*

Proof. See [14]. □

2.2 A Small Imperative Language.

In the following, we give a denotational definition of $v_\tau \in \Sigma \rightharpoonup \mathbb{O}$ for a simple imperative language. We consider a small sequential non-deterministic programming language with no procedures, no pointers and no recursion. Let $\mathcal{X}$ be a finite set of variables and let $\mathbb{I}$ be a set of values, where $\mathbb{I} \in \{\mathbb{Z}, \mathbb{Q}, \mathbb{R}\}$. In Figure 2, we define inductively the syntax of programs.

An environment $\rho \in \mathcal{X} \mapsto \mathbb{I}$ maps each variable in $\mathcal{X}$ to a value in $\mathbb{I}$. Let $\mathcal{E}$ denote the set of all environments. The semantics of an arithmetic expression A is a function $\mathcal{A}[\![A]\!] \in \mathcal{E} \mapsto \wp(\mathbb{I})$ mapping an environment to the set of all possible values for the expression in the given environment. Similarly, the semantics of a boolean expression B is a function $\mathcal{B}[\![B]\!] \in \mathcal{E} \mapsto \wp(\{\mathsf{T}, \mathsf{F}\})$ mapping an environment to the set of all possible truth values for the expression in the environment. We need power-sets because we also consider non-deterministic arithmetic and boolean expressions (cf. Figure 2). Non-determinism comes in handy to model program input and to approximate non-linear expressions.

Let $\mathcal{L}$ be a finite set of labels. The initial and final labels of a program are denoted by i and e, respectively. A state $s \in \mathcal{L} \times \mathcal{E}$ is a pair consisting of a program control point $l \in \mathcal{L}$ and an environment $\rho \in \mathcal{E}$. Let Σ denote the set of all program states. The program initial states belong to $\Sigma_i \triangleq \{s \in \Sigma \mid \exists \rho \in \mathcal{E} : s = \langle i, \rho \rangle\}$ while $\Sigma_e \triangleq \{s \in \Sigma \mid \exists \rho \in \mathcal{E} : s = \langle e, \rho \rangle\}$ is the set of program final states. The semantics of a statement S is a function $\mathcal{S}[\![S]\!] \in (\Sigma \rightharpoonup \mathbb{O}) \mapsto (\Sigma \rightharpoonup \mathbb{O})$ mapping a partial function from states to ordinals into a partial function from states to ordinals with greater value. The program semantics $v_\tau \in \Sigma \rightharpoonup \mathbb{O}$ is computed backwards, starting from the partial function $\lambda s(s \in \Sigma_e).\ 0$ and propagating it towards the initial states by means of $\mathcal{S}[\![S]\!]$.

Note that we can redefine $v_\tau \in \Sigma \rightharpoonup \mathbb{O} = (\mathcal{L} \times \mathcal{E}) \rightharpoonup \mathbb{O}$ in an isomorphic way by point-wise lifting to $\mathcal{L}$ of the partial function from environments to ordinals:

$$\mathcal{S}[\![X := A]\!]v \triangleq \lambda\rho.\ \mathsf{sup}\{v(\rho[X \mapsto n]) + 1 \mid \forall n \in \mathcal{A}[\![A]\!]\rho : \rho[X \mapsto n] \in \mathsf{dom}(v)\}$$

$$\begin{aligned}
\mathcal{S}[\![\mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}]\!]v \triangleq\ & (\lambda\rho(\rho \in \mathsf{dom}(v_1) \wedge \mathsf{F} \notin \mathcal{B}[\![B]\!]\rho).\ v_1(\rho)) \\
& \curlyvee\ (\lambda\rho(\rho \in \mathsf{dom}(v_2) \wedge \mathsf{T} \notin \mathcal{B}[\![B]\!]\rho).\ v_2(\rho))\ \curlyvee\ (v_1 \sqcup v_2) \\
& \text{where } v_1 \triangleq \lambda\rho(\rho \in \mathsf{dom}(\mathcal{S}[\![S_1]\!]v) \wedge \mathsf{T} \in \mathcal{B}[\![B]\!]\rho).\ (\mathcal{S}[\![S_1]\!]v)(\rho) + 1 \\
& \qquad v_2 \triangleq \lambda\rho(\rho \in \mathsf{dom}(\mathcal{S}[\![S_2]\!]v) \wedge \mathsf{F} \in \mathcal{B}[\![B]\!]\rho).\ (\mathcal{S}[\![S_2]\!]v)(\rho) + 1
\end{aligned}$$

$$\begin{aligned}
\mathcal{S}[\![\mathsf{while}\ B\ \mathsf{do}\ S\ \mathsf{od}]\!]v \triangleq\ & \mathsf{lfp}^{\preccurlyeq}_{\emptyset}\ \phi \\
& \text{where } \phi \triangleq \lambda x.\ (\lambda\rho(\rho \in \mathsf{dom}(v_1) \wedge \mathsf{F} \notin \mathcal{B}[\![B]\!]\rho).\ v_1(\rho)) \\
& \qquad \curlyvee\ (\lambda\rho(\rho \in \mathsf{dom}(v_2) \wedge \mathsf{T} \notin \mathcal{B}[\![B]\!]\rho).\ v_2(\rho))\ \curlyvee\ (v_1 \sqcup v_2)) \\
& \qquad v_1 \triangleq \lambda\rho(\rho \in \mathsf{dom}(\mathcal{S}[\![S_1]\!]x) \wedge \mathsf{T} \in \mathcal{B}[\![B]\!]\rho).\ (\mathcal{S}[\![S_1]\!]x)(\rho) + 1 \\
& \qquad v_2 \triangleq \lambda\rho(\rho \in \mathsf{dom}(v) \wedge \mathsf{F} \in \mathcal{B}[\![B]\!]\rho).\ v(\rho) + 1
\end{aligned}$$

$$\mathcal{S}[\![S_1; S_2]\!]v \triangleq \mathcal{S}[\![S_1]\!](\mathcal{S}[\![S_2]\!]v)$$

Fig. 3: Concrete Semantics

$v_\tau \in \mathcal{L} \mapsto (\mathcal{E} \not\to \mathbb{O})$. In a similar way, we can redefine the statement semantics: $\mathcal{S}[\![S]\!] \in (\mathcal{E} \not\to \mathbb{O}) \mapsto (\mathcal{E} \not\to \mathbb{O})$ is defined by induction on the syntax of programs in Figure 3. In this form, we can consider $v_\tau \in \mathcal{L} \mapsto (\mathcal{E} \not\to \mathbb{O})$ as an invariance semantics: to each program control point $l \in \mathcal{L}$, it associates a partial function in $\mathcal{E} \not\to \mathbb{O}$ representing the program ranking function in that particular program point. Loop semantics requires the computation of a loop invariant as the least fix-point of a monotonic function $\phi \in (\mathcal{E} \not\to \mathbb{O}) \mapsto (\mathcal{E} \not\to \mathbb{O})$. However, such a fix-point is usually not computable.

In the next section, we will present a decidable abstraction of v_τ by means of piecewise-defined functions computed through backward invariance analysis.

3 An Abstract Domain Functor for Termination

We derive an approximate program semantics by abstract interpretation [10,12]. We look for $v^\#_\tau \in \mathcal{L} \mapsto \mathcal{V}^\#$ mapping each program point $l \in \mathcal{L}$ to an abstraction of the program ranking function in that specific program point.

In particular, we abstract the ranking functions in $\mathcal{E} \not\to \mathbb{O}$ by piecewise-defined ranking functions in $\mathcal{V}^\#$. To this end, we introduce the family of segmented ranking functions abstract domains $\mathsf{V}(\mathsf{E}, \mathsf{P})$, parameterized by the environments abstract domains E and the functions abstract domains P. Adopting an OCaml terminology, each V is an abstract domain functor: a function mapping the parameter abstract domains E and P into a new abstract domain $\mathsf{V}(\mathsf{E}, \mathsf{P})$. V can be applied to various implementations of E and P yielding the corresponding implementations of $\mathsf{V}(\mathsf{E}, \mathsf{P})$, with no need for further programming effort.

In the following, in order we present the family of environments abstract domains E, the family of functions abstract domains P, and the family of pa-

rameterized abstract domains of segmented ranking function $\mathsf{V}(\mathsf{E}, \mathsf{P})$. We also describe the design of particular instances, based on intervals and affine functions, of each one of these abstract domains.

To ensure the soundness of our abstraction, throughout the rest of the paper we will continue to maintain a strict separation between approximation and computational orders (as we already did in Section 2).

3.1 Environments Abstract Domain

The environments abstract domain E abstracts sets of concrete environments in $\wp(\mathcal{E})$. The abstract properties $\rho^\# \in \mathcal{E}^\#$ are called abstract environments. The concretization function $\gamma_\mathsf{E} \in \mathcal{E}^\# \mapsto \wp(\mathcal{E})$ maps an abstract property to the set of concrete environments having that abstract property.

In case E is a non-relational domain, $\wp(\mathcal{E}) = \wp(\mathcal{X} \mapsto \mathbb{I})$ is abstracted to $\mathcal{X} \mapsto \wp(\mathbb{I})$, and we have $\mathcal{E}^\# \triangleq \mathcal{X} \mapsto \mathcal{B}^\#$, where the abstract domain B abstracts properties of values in $\mathbb{I}$ with concretization function $\gamma_\mathsf{B} \in \mathcal{B}^\# \mapsto \wp(\mathbb{I})$.

Intervals Abstract Domain. In the literature, numerous environments abstract domains have already been proposed (e.g., the numerical abstract domains of intervals [9], octagons [21], and convex polyhedra [16]).

In the following, as a simple example of non-relational environments abstract domain, we will consider the intervals abstract domain [9]. The abstract properties in $\mathcal{B}^\#$ are empty ($\bot_\mathsf{B}$) or non-empty ($[a, b]$) intervals with bounds in $\mathbb{I} \cup \{-\infty, +\infty\}$. We denote the abstract partial order by $\sqsubseteq_\mathsf{B}$, the join operator by $\sqcup_\mathsf{B}$, the meet operator by $\sqcap_\mathsf{B}$ and the widening operator by $\triangledown_\mathsf{B}$.

As for the abstract transformers for assignments and tests, we recall that our program concrete semantics is *defined* backwards (cf. Section 2), and we will see (in Section 3.4) that the program abstract semantics is *computed* backwards as well. Consequently, we consider backward assignment and test transfer functions, denoted by ASSIGN_B and FILTER_B, respectively. The primitive ASSIGN_B returns an abstraction of a set of environments that can lead to another given abstraction of a set of environments by an assignment $X := A$. The primitive FILTER_B filters out environments that do not verify a boolean expression B.

Example 2. Let us consider the ranking function at program point 2 in the second iteration column of Figure 1. The test $x \geq 0$ is applied to each segment of the function, yielding the function $2[x \geq 0]$. In particular, we consider one of the segments on which such function is defined: the segment represented by the environment $\rho^\# \equiv x \mapsto [-\infty, 2]$. The result of FILTER_B for $x \geq 0$ on $\rho^\#$ is $x \mapsto [0, 2]$.

Let us consider now the assignment $x := -2x + 10$ applied segment-wise to the ranking function at program point 1 in the last column of the table. In particular, the result of ASSIGN_B on the segment represented by the environment $x \mapsto [4, 5]$ is the segment represented by $x \mapsto [3, 3]$ (recall that we consider the space of values for the variable x to be the set of integers $\mathbb{Z}$). □

3.2 Functions Abstract Domain

The functions abstract domain P is itself a functor $\mathsf{P}(\mathsf{E})$, parameterized by the environment abstract domain E. It abstracts partial functions $\mathcal{E} \rightharpoonup \mathbb{O}$ from environments to ordinals by natural-valued partial functions of the $\mathbb{I}$-valued variables in $\mathcal{X}$. Let n denote $|\mathcal{X}|$. The abstract properties of P belong to $\mathcal{E}^\# \times \mathcal{F}^\#$, where $\mathcal{F}^\# \triangleq \{\bot_\mathsf{F}\} \cup \{f^\# \mid f^\# \in \mathbb{I}^n \mapsto \mathbb{N}\} \cup \{\top_\mathsf{F}\}$. The bottom function $\bot_\mathsf{F}$ denotes the totally undefined function, and the top function $\top_\mathsf{F}$, abstracts all functions mapping environments to infinite ordinals.

The concretization function $\gamma_\mathsf{P} \in (\mathcal{E}^\# \times \mathcal{F}^\#) \mapsto (\mathcal{E} \rightharpoonup \mathbb{O})$ depends on the value of the variables in $\mathcal{X}$ according to an abstract environment $\rho^\# \in \mathcal{E}^\#$:

$$\begin{aligned}
&\gamma_\mathsf{P}(\langle \rho^\#, \bot_\mathsf{F} \rangle) = \dot{\emptyset} \\
&\gamma_\mathsf{P}(\langle \rho^\#, f^\# \rangle) = \lambda \rho \in \gamma_\mathsf{E}(\rho^\#).\ f^\#(\rho(x_1), \ldots, \rho(x_n)) \\
&\gamma_\mathsf{P}(\langle \rho^\#, \top_\mathsf{F} \rangle) = \dot{\emptyset}
\end{aligned}$$

We define the abstract approximation preorder $\sqsubseteq_\mathsf{P}$, in such a way that $\langle \rho_1^\#, f_1^\# \rangle \sqsubseteq_\mathsf{P} \langle \rho_2^\#, f_2^\# \rangle \Leftrightarrow \gamma_\mathsf{P}(\langle \rho_1^\#, f_1^\# \rangle) \sqsubseteq \gamma_\mathsf{P}(\langle \rho_2^\#, f_2^\# \rangle)$, as follows:

$$\langle \rho_1^\#, f_1^\# \rangle \sqsubseteq_\mathsf{P} \langle \rho_2^\#, f_2^\# \rangle \triangleq \rho_2^\# \sqsubseteq_\mathsf{E} \rho_1^\# \wedge f_1^\# \sqsubseteq_\mathsf{F} f_2^\#$$

where

$$f_1^\# \sqsubseteq_\mathsf{F} f_2^\# \triangleq \forall \rho \in \gamma_\mathsf{E}(\rho_1^\# \sqcap_\mathsf{E} \rho_2^\#) : f_1^\#(\rho(x_1), \ldots, \rho(x_n)) \leq f_2^\#(\rho(x_1), \ldots, \rho(x_n)).$$

Theorem 2. $\langle \rho_1^\#, f_1^\# \rangle \sqsubseteq_\mathsf{P} \langle \rho_2^\#, f_2^\# \rangle \Leftrightarrow \gamma_\mathsf{P}(\langle \rho_1^\#, f_1^\# \rangle) \sqsubseteq \gamma_\mathsf{P}(\langle \rho_2^\#, f_2^\# \rangle)$.

The result proves that γ_P is monotonic.

We also define a computational partial order $\preccurlyeq_\mathsf{P}$:

$$\langle \rho_1^\#, f_1^\# \rangle \preccurlyeq_\mathsf{P} \langle \rho_2^\#, f_2^\# \rangle \triangleq \rho_1^\# \sqsubseteq_\mathsf{E} \rho_2^\# \wedge f_1^\# \sqsubseteq_\mathsf{F} f_2^\#.$$

Lemma 2.

$$(\rho_1^\# \sqsubseteq_\mathsf{E} \rho_2^\# \wedge \rho_2^\# \sqsubseteq_\mathsf{E} \rho_1^\#) \Rightarrow (\langle \rho_1^\#, f_1^\# \rangle \sqsubseteq \langle \rho_2^\#, f_2^\# \rangle \Leftrightarrow \langle \rho_1^\#, f_1^\# \rangle \preccurlyeq \langle \rho_2^\#, f_2^\# \rangle).$$

Finally, in addition to a join operator $\sqcup_\mathsf{P}$, P is equipped with backward assignment and test abstract transformers ASSIGN_P and FILTER_P. In the following, we will define these operators for the affine functions abstract domain.

Affine Functions Abstract Domain. As an example of functions abstract domain, we instantiate the functor P with the intervals environment abstract domain E described above, and as abstract properties $f^\# \in \mathcal{F}^\#$ we choose affine functions of the form:

$$y = f(x_1, \ldots, x_n) = m_1 x_1 + \cdots + m_n x_n + q$$

where $x_1, \ldots, x_n$ are variables in $\mathcal{X}$, $y \notin \mathcal{X}$ is a special variable not included in $\mathcal{X}$, and $m_1, \ldots, m_n, q$ are constants.

The operators of the affine functions abstract domain include the join operator $\sqcup_\mathsf{P}$, and the abstract property transformers ASSIGN_P for backward assignments and FILTER_P for backward tests.

Join. The join operator $\sqcup_{\mathsf{P}}$, given two partial functions $\langle \rho_1^\#, f_1^\# \rangle$ and $\langle \rho_2^\#, f_2^\# \rangle$, determines $\rho^\# \equiv \rho_1^\# \sqcap_{\mathsf{E}} \rho_2^\#$ and then computes $f^\# \equiv f_1^\# \sqcup_{\mathsf{F}} f_2^\#$ within $\rho^\#$.

Let $\rho^\# \equiv \{x_1 \mapsto [a_1, b_1], \ldots, x_n \mapsto [a_n, b_n]\}$, $f_1^\# \equiv y = f_1(x_1, \ldots, x_n)$ and $f_2^\# \equiv y = f_2(x_1, \ldots, x_n)$. The operator $\sqcup_{\mathsf{F}}$ basically reuses the join of polyhedra [16]; it transforms $f_1^\#$ and $f_2^\#$ into two set of constraints of the form:

$$\{a_1 \leq x_1 \leq b_1, \ldots, a_n \leq x_n \leq b_n, 0 \leq y \leq f_i(x_1, \ldots, x_n)\}$$

for $i = 1, 2$. Then, it computes their convex hull:

$$\{a_1 \leq x_1 \leq b_1, \ldots, a_n \leq x_n \leq b_n, 0 \leq y \leq f(x_1, \ldots, x_n)\}$$

and transforms it back to $\langle \rho^\#, f^\# \rangle$, where $f^\# \equiv y = f(x_1, \ldots, x_n)$. In case the convex hull contains more than one constraint on y (except for the constraint $0 \leq y$), we are in presence of several not comparable choices for $f^\#$. In such situation, we prefer a deterministic behavior for $\sqcup_{\mathsf{F}}$, and we choose $f^\# = \top_{\mathsf{F}}$.

Example 3. Let us consider the abstract functions

$$f_1^\# \equiv y = f_1(x_1, x_2) = -\frac{1}{2}x_2 + 2$$
$$f_2^\# \equiv y = f_2(x_1, x_2) = -\frac{1}{2}x_1 + 2$$

within the environment $\rho^\# \equiv \{x_1 \mapsto (-\infty, 4], x_2 \mapsto (-\infty, 4]\}$. Their join is the convex hull of the sets of constraints $\{x_1 \leq 4, x_2 \leq 4, 0 \leq y \leq -\frac{1}{2}x_2 + 2\}$ and $\{x_1 \leq 4, x_2 \leq 4, 0 \leq y \leq -\frac{1}{2}x_1 + 2\}$. Thus $f_1^\# \sqcup_{\mathsf{F}} f_2^\# = f^\#$ where $f^\# \equiv y = f(x_1, x_2) = -\frac{1}{2}x_1 - \frac{1}{2}x_2 + 4$ (see Figure 4). □

In the particular case where $f_1^\# \equiv \bot_{\mathsf{F}}$ or $f_2^\# \equiv \bot_{\mathsf{F}}$, their join $f_1^\# \sqcup_{\mathsf{F}} f_2^\#$ is $f^\# \equiv \bot_{\mathsf{F}}$. In all the other cases, $f_1^\# \sqcup_{\mathsf{F}} f_2^\#$ is $f^\# \equiv \top_{\mathsf{F}}$.

The following result proves the soundness of the join operator $\sqcup_{\mathsf{P}}$.

Theorem 3. $\gamma_{\mathsf{P}}(\langle \rho_1^\#, f_1^\# \rangle) \sqcup \gamma_{\mathsf{P}}(\langle \rho_2^\#, f_2^\# \rangle) \sqsubseteq \gamma_{\mathsf{P}}(\langle \rho_1^\#, f_1^\# \rangle \sqcup_{\mathsf{P}} \langle \rho_2^\#, f_2^\# \rangle)$.

Assignments. In order to handle assignments $X := A$, the abstract domain P is equipped with an operation to substitute an arithmetic expression A for a variable X within an abstract function $f^\#$. Given $\langle \rho^\#, f^\# \rangle \in \mathcal{E}^\# \times \mathcal{F}^\#$, the backward abstract transformer $\text{ASSIGN}_{\mathsf{P}}$, applies the assignment independently to $\rho^\#$, by means of $\text{ASSIGN}_{\mathsf{E}}$, and to $f^\#$. Let $f^\# \equiv f(x_1, \ldots, X, \ldots, x_n)$. The transformer $\text{ASSIGN}_{\mathsf{F}}$ has to take into account the assignment $X := A$ and increase the value of $f^\#$: the result is the function $f(x_1, \ldots, A, \ldots, x_n) + 1$.

Example 4. Let consider again the ranking function at program point 1 in the last column of Figure 1. The result of the assignment $x := -2x + 10$, on the segment represented by the environment $x \mapsto [4, 5]$ and the function $f(x) = 7$, is represented by $\rho^\# \equiv x \mapsto [3, 3]$ and $f^\# \equiv f(-2x + 10) = 7 + 1 = 8$. □

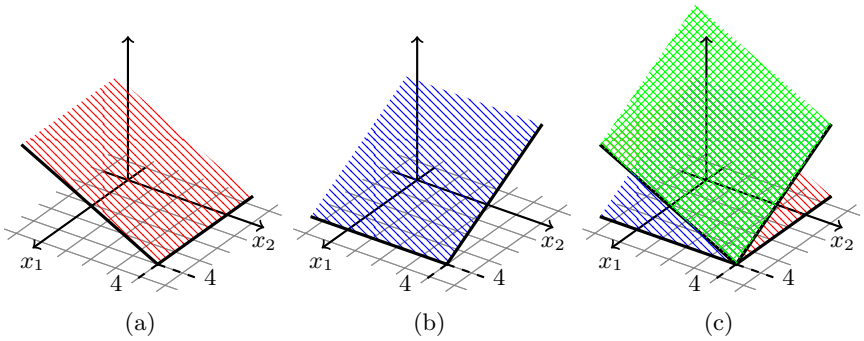


Fig. 4: Example of join of two abstract functions of two variables. The function $f_1(x_1, x_2) = -\frac{1}{2}x_2 + 2$ (shown in (a)) is joined with $f_2(x_1, x_2) = -\frac{1}{2}x_1 + 2$ (shown in (b)), within the environment $\{x_1 \mapsto (-\infty, 4], x_2 \mapsto (-\infty, 4]\}$. The result is the function $f(x_1, x_2) = -\frac{1}{2}x_1 - \frac{1}{2}x_2 + 4$ (shown in (c)).

Example 5. Let us consider $f(x) = 2x + 1$ within the environment $x \mapsto [4, 6]$, and the assignment $x := x + 1$. The result of the assignment is $\langle \rho^\#, f^\# \rangle$, where $\rho^\# \equiv x \mapsto [3, 5]$ and $f^\# \equiv f(x + 1) + 1 = 2(x + 1) + 1 + 1 = 2x + 4$. □

In case of a non-linear expression A, the limited expressiveness of the domain forces the assignment to be approximated using non-determinism and taking into account all possible outcomes of the resulting non-deterministic assignment.

Note that the assignment abstract transformer $\text{ASSIGN}_{\mathsf{P}}$ is *not sound* due to the over-approximation introduced by the environments transformer $\text{ASSIGN}_{\mathsf{E}}$.

Example 6. Let us consider $\rho^\# \equiv x \mapsto [2, 3]$ and $f^\# \equiv f(x) = x + 1$, and the assignment $x := x + [1, 2]$. The result of the assignment is $\langle \bar{\rho}^\#, \bar{f}^\# \rangle$, where $\bar{\rho}^\# \equiv x \mapsto [0, 2]$ and $\bar{f}^\# \equiv f(x + [1, 2]) \equiv \bar{f}(x) = x + 4$. It is not sound because $\mathcal{S}[\![x := x + [1, 2]]\!]\gamma_{\mathsf{P}}(\langle \rho^\#, f^\# \rangle) \not\sqsubseteq \gamma_{\mathsf{P}}(\langle \bar{\rho}^\#, \bar{f}^\# \rangle)$: in fact, the domain of $\gamma_{\mathsf{P}}(\langle \bar{\rho}^\#, \bar{f}^\# \rangle)$, that is $\{x \mapsto 0, x \mapsto 1, x \mapsto 2\}$, is not included in the domain of $\mathcal{S}[\![x := x + [1, 2]]\!]\gamma_{\mathsf{P}}(\langle \rho^\#, f^\# \rangle)$, that is $\{x \mapsto 1\}$. □

However, in the next section, we will exploit $\text{ASSIGN}_{\mathsf{P}}$ to define $\text{ASSIGN}_{\mathsf{V}}$, for the abstract domain $\mathsf{V}(\mathsf{E}, \mathsf{P})$, and we will prove the soundness of such transformer, despite the fact that it uses an unsound $\text{ASSIGN}_{\mathsf{P}}$.

Tests. The test abstract transformer $\text{FILTER}_{\mathsf{P}}$, given $\langle \rho^\#, f^\# \rangle \in \mathcal{E}^\# \times \mathcal{F}^\#$, simply narrows the domain of $f^\#$, represented by the environment $\rho^\#$, by means of the environments transformer $\text{FILTER}_{\mathsf{E}}$.

3.3 Segmented Ranking Functions Abstract Domain

The segmented ranking functions abstract domain $\mathsf{V}(\mathsf{E}, \mathsf{P})$ introduces segmentations into P: it abstracts ranking functions in $\mathcal{E} \not\to \mathbb{O}$ by piecewise-defined

abstract ranking functions belonging to:

$$\mathcal{V}^\# \triangleq \{(\mathcal{E}^\# \times \mathcal{F}^\#)^k \mid k \geq 0\}.$$

An abstract property $v^\# \in \mathcal{V}^\#$ has the form $v^\# \equiv \langle \rho_1^\#, f_1^\# \rangle \dots \langle \rho_k^\#, f_k^\# \rangle$, where $\rho_1^\#, \dots, \rho_k^\#$ are non-overlapping abstract environments forming a partition of the space of values for the program variables in $\mathcal{X}$.

Let $\bot_{\mathsf{V}}$ denote the totally undefined function.

The concretization function $\gamma_{\mathsf{V}} \in \mathcal{V}^\# \mapsto (\mathcal{E} \not\mapsto \mathbb{O})$ is defined as follows[3]:

$$\gamma_{\mathsf{V}}(v^\#) = \gamma_{\mathsf{V}}(\langle \rho_1^\#, f_1^\# \rangle \dots \langle \rho_k^\#, f_k^\# \rangle) = \dot{\bigcup_i} \gamma_{\mathsf{P}}(\langle \rho_i^\#, f_i^\# \rangle)$$

As in [15], the abstract domain $\mathsf{V}(\mathsf{E}, \mathsf{P})$ relies on a segmentation unification algorithm: given two functions $v_1^\#$ and $v_2^\#$, it modifies their segments so that they form a common refined partition of the space of values for each program variable. The abstract order $\sqsubseteq_{\mathsf{V}}$ applies such segmentation unification and then compares the abstract ranking functions. First, their domains are compared considering the number of segments $\langle \rho^\#, f^\# \rangle$ in which each of the functions is defined (i.e. in which $f^\# \neq \bot_{\mathsf{F}}$ and $f^\# \neq \top_{\mathsf{F}}$). Then, if $v_2^\#$ is defined on less segments than $v_1^\#$, the functions are compared piecewise using the functions order $\sqsubseteq_{\mathsf{P}}$.

Theorem 4. $v_1^\# \sqsubseteq_{\mathsf{V}} v_2^\# \Leftrightarrow \gamma_{\mathsf{V}}(v_1^\#) \sqsubseteq \gamma_{\mathsf{V}}(v_2^\#)$

The result shows that γ_{V} is monotonic.

We define as well a computational partial order $\preccurlyeq_{\mathsf{V}}$ that also exploits the segmentation unification algorithm. Then, it compares the domains of the functions $v_1^\#$ and $v_2^\#$ (as $\sqsubseteq_{\mathsf{V}}$ does) and, if $v_1^\#$ is defined on less segments than $v_2^\#$, it compares the functions piecewise using the partial order $\preccurlyeq_{\mathsf{P}}$.

Note that, *the approximation order $\sqsubseteq_{\mathsf{V}}$ and the computational order $\preccurlyeq_{\mathsf{V}}$ coincide when the functions are defined on the same segments.*

Let $\sqcup_{\mathsf{V}}$ denote the join operator, $\triangledown_{\mathsf{V}}$ the widening operator, and let $\text{ASSIGN}_{\mathsf{V}}$ and $\text{FILTER}_{\mathsf{V}}$ denote the backward assignment and test transfer functions, respectively. In the following, we will define these operators and prove their soundness for an instance of $\mathsf{V}(\mathsf{E}, \mathsf{P})$ with intervals and affine functions.

Segmented Affine Ranking Functions Abstract Domain As an example of segmented ranking functions abstract domain, we apply the functor V to the interval environments abstract domain E (described in Section 3.1) and to the affine functions abstract domain $\mathsf{P}(\mathsf{E})$ (described in Section 3.2). The abstract properties $v^\# \in \mathcal{V}^\#$ are piecewise-defined affine ranking functions.

In this case, since the segments are determined by abstract intervals with constant bounds, the segmentation unification algorithm is rather simple: the unification simply introduces new bounds consequently splitting intervals in both segmentations. An example of segmentation unification is illustrated in Figure 5.

[3] $\dot{\cup}$ joins partial functions with disjoint domains: $(f_1 \dot{\cup} f_2)(x) \triangleq f_1(x)$, if $x \in \mathsf{dom}(f_1)$, and $(f_1 \dot{\cup} f_2)(x) \triangleq f_2(x)$, if $x \in \mathsf{dom}(f_2)$, where $\mathsf{dom}(f_1) \cap \mathsf{dom}(f_2) = \emptyset$.

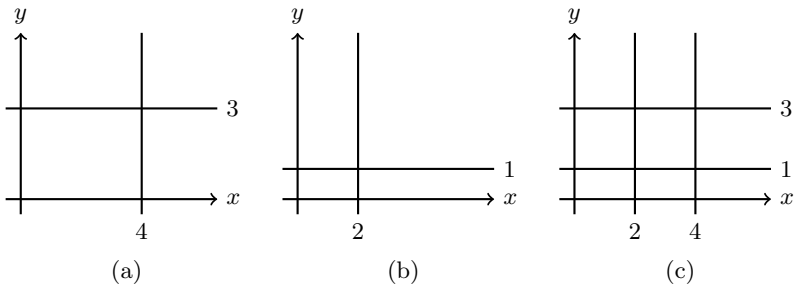


Fig. 5: Example of segmentation unification. The segmentation shown in (a) is joined with the one shown in (b). The resulting segmentation is depicted in (c).

Join. As the order $\sqsubseteq_{\mathsf{V}}$, also the join operator $\sqcup_{\mathsf{V}}$ depends on the segmentation unification algorithm. After the unification, the abstract ranking functions are joined piecewise by means of the abstract functions join operator $\sqcup_{\mathsf{P}}$.

The next result proves the soundness of $\sqcup_{\mathsf{V}}$.

Theorem 5. $\gamma_{\mathsf{V}}(v_1^{\#}) \sqcup \gamma_{\mathsf{V}}(v_2^{\#}) \sqsubseteq \gamma_{\mathsf{V}}(v_1^{\#} \sqcup_{\mathsf{V}} v_2^{\#})$

We now define another join operator $\curlyvee_{\mathsf{V}}$ that we will use in the following to join two functions $v_1^{\#}$ and $v_2^{\#}$, the segmentations of which have different lower and upper bounds. Where both segmentation are defined, $\curlyvee_{\mathsf{V}}$ applies the segmentation unification algorithm and then joins the ranking functions piecewise using the join operator $\sqcup_{\mathsf{P}}$. To the resulting segmented function, segments are added, where only one of the functions is defined.

Example 7. Let us consider the abstract piecewise-defined ranking functions $v_1^{\#} \equiv \langle x \mapsto [2,4], y = 2\rangle$ and $v_2^{\#} \equiv \langle x \mapsto (-\infty,4], y = -x+4\rangle$ represented in Figure 6a and Figure 6b, respectively. Their join is the piecewise-defined ranking function $v^{\#} \equiv \langle x \mapsto (-\infty,2), y = -x+4\rangle\langle x \mapsto [2,4], y = 2\rangle$ of Figure 6c. □

Assignments. The backward assignment abstract transformer $\text{ASSIGN}_{\mathsf{V}}$, given a segmented function $v^{\#}$, applies piecewise the transformer $\text{ASSIGN}_{\mathsf{P}}$ and then joins the resulting segments using the join operator $\curlyvee_{\mathsf{V}}$. In this way, it refines the segmentation of the function so as to avoid overlapping segments.

Example 8. Let us consider $v^{\#} \equiv \langle x \mapsto [-\infty,9], \bot_{\mathsf{F}}\rangle\langle x \mapsto [10,+\infty], y = 0\rangle$ and the assignment $x := x + [0,2]$. The result of the assignment on $v^{\#}$ is $\langle x \mapsto [-\infty,9], \bot_{\mathsf{F}}\rangle \curlyvee_{\mathsf{V}} \langle x \mapsto [8,+\infty], y = 1\rangle$. That is the segmented function $\langle x \mapsto [-\infty,7], \bot_{\mathsf{F}}\rangle\langle x \mapsto [8,9], \bot_{\mathsf{F}}\rangle\langle x \mapsto [10,+\infty], y = 1\rangle$.

The following result proves the soundness of $\text{ASSIGN}_{\mathsf{V}}$.

Theorem 6. $\mathcal{S}[\![X := A]\!]\gamma_{\mathsf{V}}(v^{\#}) \sqsubseteq \gamma_{\mathsf{V}}(\text{ASSIGN}_{\mathsf{V}}(X := A, v^{\#}))$.

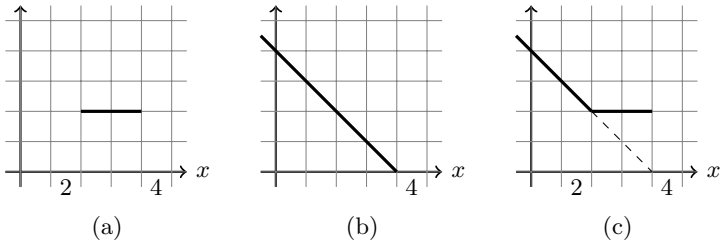


Fig. 6: Example of join of partial piecewise-defined ranking functions. $v_1^\#$ (shown in (a)) is joined with $v_2^\#$ (shown in (b)). The result $v^\#$ is shown in (c).

We omit the proof due to space limits. Intuitively, as we have seen in the previous section, the transformer ASSIGN_P is unsound because it introduces over-approximations. In particular, over-approximations are more likely to appear because of non-determinism (cf. Example 8). However, since the resulting segments are joined by means of $\curlyvee_\mathsf{V}$, we recover from the unsoundness of ASSIGN_P. In fact, by definition of $\curlyvee_\mathsf{V}$, the possible overlaps are handled with the sound join operator $\sqcup_\mathsf{P}$ and this yields a sound backward assignment transformer ASSIGN_V.

Tests. The transformer FILTER_V for backward tests simply applies piecewise the transformer FILTER_P.

In the following (cf. Figure 8), we will exploit FILTER_V and the operator $\curlyvee_\mathsf{V}$ to define the abstract counterpart of the concrete semantics for the `if` statement, and the abstract counterpart $\phi^\# \in \mathcal{V}^\# \mapsto \mathcal{V}^\#$ of $\phi \in (\mathcal{E} \rightharpoonup \mathbb{O}) \mapsto (\mathcal{E} \rightharpoonup \mathbb{O})$, as defined in Figure 3 for the `while` statement. The soundness of these operators relies on an argument similar to the one we used to justify ASSIGN_V.

Widening. The widening operator $\triangledown_\mathsf{V}$ prevents the number of pieces of an abstract ranking function from growing indefinitely. First, to avoid infinite chains, it performs a segmentation unification that keeps only the bounds occurring in the first segmentation. Then, it widens the functions piecewise (basically reusing[4] the widening on polyhedra) and toward the adjacent segments (cf. Example 9).

Example 9. Let us consider the widening between the segmented functions $v_1^\#$ and $v_2^\#$ represented in Figure 7a and Figure 7b, respectively. The operator $\triangledown_\mathsf{V}$ keeps only the segmentation of $v_1^\#$. Thus, the segments $\langle x \mapsto (-\infty, 3), \bot_\mathsf{F} \rangle$ and $\langle x \mapsto [3, 5), y = 5 \rangle$ of $v_2^\#$, are joined into a single segment $\langle x \mapsto (-\infty, 5), y = 5 \rangle$. Then, $\langle x \mapsto (-\infty, 5), y = 5 \rangle$ is widened toward $\langle x \mapsto [5, 10), y = 3 \rangle$: both segments are considered as sets of constraints (as we have seen for the definition of the operator $\sqcup_\mathsf{F}$ of Section 3.2) and their convex-hull

$$\{x \leq 10, 0 \leq y, y \leq 3, y \leq -\frac{2}{5}x + 7\}$$

[4] In a similar way as the join of polyhedra was reused to define $\sqcup_\mathsf{F}$ in Section 3.2.

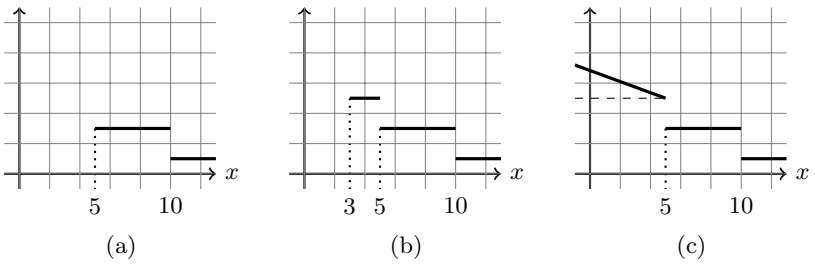


Fig. 7: Example of widening of abstract piecewise-defined ranking functions. The result of widening $v_1^\#$ (shown in (a)) with $v_2^\#$ (shown in (b) is shown in (c).

is shrunk by the constraint $x < 5$ originating from the first segment:

$$\{x < 5, 0 \leq y \leq -\frac{2}{5}x + 7\}.$$

The resulting widened segmented function is

$$v^\# \equiv \langle x \mapsto (-\infty, 5), y = -\frac{2}{5}x + 7\rangle\langle x \mapsto [5, 10), y = 3\rangle\langle x \mapsto [10, +\infty), y = 1\rangle$$

represented in Figure 7c. □

Note that this widening does not respect the traditional definition [10], since the property $\gamma_{\mathsf{V}}(v_1^\#) \sqcup \gamma_{\mathsf{V}}(v_2^\#) \sqsubseteq \gamma_{\mathsf{V}}(v_1^\# \triangledown_{\mathsf{V}} v_2^\#)$ does not always hold.

However, we are able to prove the following weaker result (that will be decisive for the soundness of the iterations with widening):

Lemma 3. $(X \triangledown_{\mathsf{V}} Y = X) \Rightarrow Y \sqsubseteq_{\mathsf{V}} X$

Proof. When $X \triangledown_{\mathsf{V}} Y = X$, we have $Y \preccurlyeq_{\mathsf{V}} X$. Moreover, since the widening force the segmentation of X on Y, having $X \triangledown_{\mathsf{V}} Y = X$ means that X and Y are defined on the same segments. In this case, as we have already observed, the orders $\sqsubseteq_{\mathsf{V}}$ and $\preccurlyeq_{\mathsf{V}}$ coincide, and we have $Y \sqsubseteq_{\mathsf{V}} X$. □

3.4 Abstract Termination Semantics

We now use the operators of $\mathsf{V}(\mathsf{E}, \mathsf{P})$ to define the statement abstract semantics $S^\#[\![S]\!] \in V^\# \mapsto V^\#$ by induction on the syntax of programs in Figure 8.

The program abstract semantics $r_\tau^\# \in \mathcal{L} \mapsto \mathcal{V}^\#$ is computed through backward invariance analysis, starting from the program final control point $e \in \mathcal{L}$ with the constant function equals to 0. The ranking function is then propagated towards the program initial control point $i \in \mathcal{L}$ taking assignments and tests

$$\mathcal{S}^\#[\![X := A]\!]v \triangleq \text{ASSIGN}_{\mathsf{V}}(X := A, v)$$
$$\mathcal{S}^\#[\![\mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}]\!]v \triangleq \text{FILTER}_{\mathsf{S}}(B, \mathcal{S}^\#[\![S_1]\!]v)\ \curlyvee_{\mathsf{V}}\ \text{FILTER}_{\mathsf{S}}(\neg B, \mathcal{S}^\#[\![S_2]\!]v)$$
$$\mathcal{S}^\#[\![\mathsf{while}\ B\ \mathsf{do}\ S\ \mathsf{od}]\!]v \triangleq \mathsf{lfp}^{\#\preccurlyeq_{\mathsf{V}}}_{\bot_{\mathsf{V}}}\ \phi^\#$$
$$\text{where } \phi^\# \triangleq \lambda x.\ \text{FILTER}_{\mathsf{S}}(\neg B, v)\ \curlyvee_{\mathsf{V}}\ \text{FILTER}_{\mathsf{S}}(B, \mathcal{S}^\#[\![S]\!]x)$$
$$\mathcal{S}^\#[\![S_1; S_2]\!]v \triangleq \mathcal{S}^\#[\![S_1]\!](\mathcal{S}^\#[\![S_2]\!]v)$$

Fig. 8: Abstract Semantics

into account with widening around loops [4]. The upward iteration sequence with widening

$$X_0 = \bot_{\mathsf{V}}$$
$$X_{i+1} = X_i\ \triangledown_{\mathsf{V}}\ \phi^\#(X_i)$$

is ultimately stationary and we prove that its limit $\mathsf{lfp}^{\#\preccurlyeq_{\mathsf{V}}}_{\bot_{\mathsf{V}}}\ \phi^\#$ is a sound over-approximation of $\mathsf{lfp}^{\preccurlyeq}_{\emptyset}\ \phi$:

Lemma 4. $\mathsf{lfp}^{\preccurlyeq}_{\emptyset}\ \phi \sqsubseteq \gamma_{\mathsf{V}}(\mathsf{lfp}^{\#\preccurlyeq_{\mathsf{V}}}_{\bot_{\mathsf{V}}}\ \phi^\#)$

Proof. Follows from the soundness of the function $\phi^\#$ and Lemma 3. □

Finally, thanks to the soundness of all abstract operators, with the following result we establish the soundness of the program semantics $r^\#_\tau$ for proving program termination for initial states in I.

Theorem 7. *Let $v^\#$ be such that $r^\#_\tau \sqsubseteq_{\mathsf{V}} v^\# \wedge I \subseteq \mathsf{dom}(\gamma_{\mathsf{V}}(v^\#))$. Then, a program P, with trace semantics described by a transition system $\langle \Sigma, \tau \rangle$, terminates for all traces starting from initial states in $I \in \wp(\Sigma)$.*

4 Implementation

We have implemented a research prototype static analyzer, based on our abstract domain of segmented ranking functions. It is written in OCaml on top of the Apron library [20], and we have used it to analyze programs written in the small non-deterministic imperative language presented in Section 2.2.

To improve precision, we avoid trying to compute a ranking function for the non-reachable states: our tool runs an iterated forward and backward invariance analysis to over-approximate the reachable states definitely leading to final states [11]. Then, it runs the backward analysis to infer the 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 until an abstract fix-point is reached. In case of nested loops, a fix-point on the inner loop is computed for each iteration of the outer loop, following [4].

To illustrate the expressiveness of our domain, we consider more examples, besides the one shown in Section 1.

Example 10. Let us consider the following program:

$$
\begin{array}{l}
\mathsf{while}\ {}^{1}(x_1 \geq 0 \wedge x_2 \geq 0)\ \mathsf{do} \\
\quad \mathsf{if}\ {}^{2}(?)\ \mathsf{then} \\
\quad\quad {}^{3}x_1 := x_1 - 1 \\
\quad \mathsf{else} \\
\quad\quad {}^{4}x_2 := x_2 - 1 \\
\quad \mathsf{fi} \\
\mathsf{od}^{5}
\end{array}
$$

The presence of the test within the loop does not impair our method.

We run our analysis delaying widening of 2 iterations, and we obtained the following loop invariant at program point 1:

$$
f(x_1, x_2) = \begin{cases}
1 & x_1 < 0 \\
1 & x_2 < 0 \\
3 & 0 \leq x_1 < 1 \wedge 0 \leq x_2 < 1 \\
5 & 0 \leq x_1 < 1 \wedge 1 \leq x_2 < 2 \\
5 & 1 \leq x_1 < 2 \wedge 0 \leq x_2 < 1 \\
2x_1 + 3 & 2 \leq x_1 \wedge 0 \leq x_2 < 1 \\
2x_2 + 3 & 0 \leq x_1 < 1 \wedge 2 \leq x_2 \\
2x_1 + 2x_2 + 3 & 1 \leq x_1 \wedge 1 \leq x_2
\end{cases}
$$

Note how the ranking function, since its value represents an upper bound on the number of steps to termination, also provides information on the program computational complexity. □

Example 11. Let us consider the following program:

$$
\begin{array}{l}
\mathsf{while}\ {}^{1}(x < 10)\ \mathsf{do} \\
\quad {}^{2}x := 2x \\
\mathsf{od}^{3}
\end{array}
$$

Such program always terminates if and only if $x > 0$.

Our tool, with delayed widening of 2 iterations, is able to provide the following loop invariant:

$$
f(x) = \begin{cases}
3 & 5 \leq x < 10 \\
1 & 10 \leq x
\end{cases}
$$

We can see that even when the analysis fails to prove whole program termination, it can still infer useful sufficient conditions for termination.

Besides, in this case, if we assume that the variable x takes values in $\mathbb{Z}$, it is sufficient to further delay the widening, to obtain the most precise ranking function:

$$f(x) = \begin{cases} 9 & x = 1 \\ 7 & x = 2 \\ 5 & 3 \leq x \leq 4 \\ 3 & 5 \leq x \leq 9 \\ 1 & 10 \leq x \end{cases}$$

□

5 Related Work

Termination analysis has attracted increased interest over the years (cf. [8]). Proving termination of programs is necessary for ensuring the correct behavior of systems, especially those for which unexpected behavior can be catastrophic.

The first results in this field date back to [26] and [18]. In the recent past, despite the undecidability of termination, termination analysis has benefited from many research advances and powerful termination provers have emerged.

Many results are developed on the basis of the transition invariants method introduced in [23]. In particular, the Terminator prover [7] is based on an algorithm for the iterative construction of transition invariants. This algorithm search for counterexamples (i.e. single paths of a program) to termination, computes a ranking function for each one of them individually (as in [22]), and combines them into a single termination argument. Our approach differs in that it aims to prove termination for all program paths at the same time, without resorting to counterexample-guided analysis. Moreover, it avoids the cost of explicit checking for the well-foundedness of the termination argument. The approach presented in [25] shares similar motivations, but prefers loop summarization to iterative fix-point computation with widening, as considered in this paper.

Among the methods based on transition invariants, we also recall the strategy, proposed in [3], based on the *indirect* use of invariants to prove program termination (and liveness properties). On the other hand, our approach infers ranking functions *directly* as invariants.

In [2], the invariants are pre-computed as in [3], but each program point is assigned with a ranking function (that also provides information on the program computational complexity), as in our technique.

Finally, in the literature, we found only few works that have addressed the problem of automatically finding preconditions to program termination. In [6], the authors proposed a method based on preconditions generating valid ranking functions. Our approach somehow goes the other way around, using the computation of ranking functions to infer sufficient condition for termination.

6 Conclusions and Future Work

In this paper, we presented a family of parameterized abstract domains for proving termination of imperative programs. These domains automatically synthesize piecewise-defined ranking functions through backward invariance analysis.

We also described the design and implementation of a particular instance of these generic abstract domains based on intervals and affine functions. We have seen that the piecewise-definition of the functions allows us to overcome the non-existence of a linear ranking function for a program (cf. Section 1). Our invariance analysis is not limited to simple loop (cf. Example 10) and, even when it fails to prove whole program termination, it can still infer useful information as sufficient conditions for termination (cf. Example 11).

As might be expected, the implemented domain has a limited expressiveness that translates into an imprecision of the analysis especially in the case of nested loops (and, in general, of programs with non-linear complexity). For this reason, we would like to consider the possibility of structuring computations as suggested by [14]. It also remains for future work to design more abstract domains, based on non-linear functions as exponentials [17] or polynomials. In addition, we plan to extend our research to proving other liveness properties.

We are interested as well in exploring further the possible potential of our approach in the termination-related field of automatic cost analysis [1].

Finally, another line of research would be proving definite non-termination by abstraction of the potential termination semantics proposed in [14].

Acknowledgments. We are deeply grateful to Patrick Cousot, Radhia Cousot, Antoine Miné, Xavier Rival, Jérôme Feret, Damien Massé and the anonymous referees for all their useful comments and helpful suggestions.

References

1. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *J. Autom. Reasoning*, 46(2):161–203, 2011.
2. C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-Dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *SAS*, pages 117–133, 2010.
3. J. Berdine, A. Chawdhary, B. Cook, D. Distefano, and P. W. O'Hearn. Variance Analyses from Invariance Analyses. In *POPL*, pages 211–224, 2007.
4. F. Bourdoncle. Efficient Chaotic Iteration Strategies with Widenings. In *FMPA*, pages 128–141, 1993.
5. A. R. Bradley, Z. Manna, and H. B. Sipma. The Polyranking Principle. In *ICALP*, pages 1349–1361, 2005.
6. B. Cook, S. Gulwani, T. Lev-Ami, A. Rybalchenko, and M. Sagiv. Proving Conditional Termination. In *CAV*, pages 328–340, 2008.
7. B. Cook, A. Podelski, and A. Rybalchenko. Terminator: Beyond Safety. In *CAV*, pages 415–418, 2006.
8. B. Cook, A. Podelski, and A. Rybalchenko. Proving Program Termination. *Commun. ACM*, 54(5):88–98, 2011.

9. P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130, 1976.
10. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, pages 238–252, 1977.
11. P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *J. Log. Program.*, 13(2&3):103–179, 1992.
12. P. Cousot and R. Cousot. Abstract Interpretation Frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.
13. P. Cousot and R. Cousot. Higher Order Abstract Interpretation (and Application to Comportment Analysis Generalizing Strictness, Termination, Projection, and PER Analysis. In *ICCL*, pages 95–112, 1994.
14. P. Cousot and R. Cousot. An Abstract Interpretation Framework for Termination. In *POPL*, pages 245–258, 2012.
15. P. Cousot, R. Cousot, and F. Logozzo. A Parametric Segmentation Functor for Fully Automatic and Scalable Array Content Analysis. In *POPL*, pages 105–118, 2011.
16. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *POPL*, pages 84–96, 1978.
17. J. Feret. The Arithmetic-Geometric Progression Abstract Domain. In *VMCAI*, pages 42–58, 2005.
18. R. W. Floyd. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.
19. B. Jeannet. Dynamic Partitioning in Linear Relation Analysis: Application to the Verification of Reactive Systems. *Formal Methods in System Design*, 23(1):5–37, 2003.
20. B. Jeannet and A. Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *CAV*, pages 661–667, 2009.
21. A. Miné. The Octagon Abstract Domain. *HOSC*, 19(1):31–100, 2006.
22. A. Podelski and A. Rybalchenko. A Complete Method for the Synthesis of Linear Ranking Functions. In *VMCAI*, pages 239–251, 2004.
23. A. Podelski and A. Rybalchenko. Transition Invariants. In *LICS*, pages 32–41, 2004.
24. X. Rival and L. Mauborgne. The Trace Partitioning Abstract Domain. *ACM Transactions on Programming Languages and Systems*, 29(5), 2007.
25. A. Tsitovich, N. Sharygina, C. M. Wintersteiger, and D. Kroening. Loop Summarization and Termination Analysis. In *TACAS*, pages 81–95, 2011.
26. A. Turing. Checking a Large Routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, 1948.