Inferring Sufficient Conditions with Backward Polyhedral Under-Approximations

Antoine Miné[1,2]

CNRS & École Normale Supérieure
45, rue d'Ulm, 75005 Paris
France

Abstract

In this article, we discuss the automatic inference of sufficient pre-conditions by abstract interpretation and sketch the construction of an under-approximating backward analysis. We focus on numeric domains and propose transfer functions, including a lower widening, for polyhedra, without resorting to disjunctive completion nor complementation, while soundly handling non-determinism. Applications include the derivation of sufficient conditions for a program to never step outside an envelope of safe states, or dually to force it to eventually fail. Our construction is preliminary and essentially untried, but we hope to convince that this avenue of research is worth considering.

Keywords: abstract interpretation, static analysis, polyhedra, backward analysis.

1 Introduction

A major problem studied in program verification is the automatic inference of invariants and necessary conditions for programs to be correct. In this article, we consider a related problem: the inference of *sufficient conditions*.

Consider the simple loop in Fig. 1, where `j` is incremented by a random value in $[0; 1]$ at each iteration. A forward invariant analysis would find that, at the end of the loop, $\texttt{j} \in [0; 110]$ and the assertion can be violated. A backward analysis of necessary conditions would not infer any new condition on the initial value of `j` because any value in $[0; 10]$ has an execution satisfying the assertion. However, a backward sufficient condition analysis would infer

[1] This work is supported by the INRIA project "Abstraction" common to CNRS and ENS in France.
[2] Email: mine@di.ens.fr

```
j = [0;10]; i = 0;
while (i < 100) { i++; j = j + [0;1]; }
assert (j <= 105);
```

Fig. 1. Simple loop example.

that, for the assertion to always hold, it is sufficient to start with $\mathtt{j} \in [0;5]$. Applications of sufficient conditions include: counter-example generation [3], contract inference [6], verification driven by temporal properties [12], optimizing compilation by hoisting safety checks, etc.

Abstract interpretation [4] has been applied with some success [1] to the automatic generation of (over-approximated) invariants, thanks notably to the design of effective abstract domains, in particular numeric domains [7], allowing efficient symbolic computations in domains of infinite size and height. Yet, it has barely been applied to the automatic inference of sufficient conditions (although [4] discusses under-approximations) and then generally using finite domains or bounded control-flow paths [3], while logic-based weakest precondition methods [8] have thrived. We attribute this lack to the perceived difficulty in designing theoretically optimal [17] as well as practical under-approximations, and the fact that sufficient and necessary conditions differ in the presence of non-determinism. Existing solutions are restricted to deterministic programs, exact abstract domains (e.g., disjunctive completions, which do not scale well), or set-complements of over-approximating domains (e.g., disjunctions of linear inequalities, that cannot express invariants as simple as $\mathtt{j} \in [0;5]$). We present here a preliminary work that hints towards the opposite: it seems possible to define reasonably practical (although non-optimal) polyhedral abstract under-approximations for non-deterministic programs.

Section 2 introduces sufficient conditions at the level of transition systems. Section 3 presents some algebraic properties of backward functions, which are exploited in Sec. 4 to design under-approximated operators for polyhedra. Section 5 discusses related work and Sec. 6 concludes.

2 Transition Systems

2.1 Invariants and sufficient conditions

To stay general, we consider, following [4], a small-step operational semantics and model programs as transition systems (Σ, τ); Σ is a set of states and $\tau \subseteq \Sigma \times \Sigma$ is a transition relation. An execution trace is a finite or infinite countable sequence of states $(\sigma_1, \ldots, \sigma_i, \ldots) \in \Sigma^\infty$ such that $\forall i : (\sigma_i, \sigma_{i+1}) \in \tau$.

Invariants. The invariant inference problem consists in, given a set $I \subseteq \Sigma$ of initial states, inferring the set $\mathrm{inv}(I)$ of states encountered in all executions

starting in I. This set can be expressed as a fixpoint following Cousot [4]: [3]

$$\mathrm{inv}(I) = \mathrm{lfp}_I\, \lambda X.X \cup \mathrm{post}(X) \tag{1}$$

where $\mathrm{lfp}_x\, f$ is the least fixpoint of f greater than or equal to x and $\mathrm{post}(X) \stackrel{\text{def}}{=} \{\, \sigma \in \Sigma \mid \exists \sigma' \in X : (\sigma', \sigma) \in \tau \,\}$.

Sufficient conditions. In this article, we consider the reverse problem: sufficient condition inference, which consists in, given an invariant set T to obey, inferring the set of initial states $\mathrm{cond}(T)$ that guarantee that all executions stay in T. It is also given in fixpoint form following Bourdoncle [2]: [4]

$$\mathrm{cond}(T) = \mathrm{gfp}_T\, \lambda X.X \cap \widetilde{\mathrm{pre}}(X) \tag{2}$$

where $\mathrm{gfp}_x\, f$ is the greatest fixpoint of f smaller than or equal to x and $\widetilde{\mathrm{pre}}(X) \stackrel{\text{def}}{=} \{\, \sigma \in \Sigma \mid \forall \sigma' \in \Sigma : (\sigma, \sigma') \in \tau \implies \sigma' \in X \,\}$. $\mathrm{cond}(T)$ is indeed a sufficient condition and, in fact, the most general sufficient condition:

Theorem 2.1 $\forall T, X : \mathrm{inv}(\mathrm{cond}(T)) \subseteq T$ *and* $\mathrm{inv}(X) \subseteq T \implies X \subseteq \mathrm{cond}(T)$.

Non-determinism. The function $\widetilde{\mathrm{pre}}$ we use differs from the function pre used in most backward analyses [2,4,16] and defined as $\mathrm{pre}(X) \stackrel{\text{def}}{=} \{\, \sigma \in \Sigma \mid \exists \sigma' \in X : (\sigma, \sigma') \in \tau \,\}$. Indeed, $\widetilde{\mathrm{pre}}(X) \neq \mathrm{pre}(X)$ when the transition system is non-deterministic, i.e., some states have several successors or none. Non-determinism is useful to model unspecified parts of programs, such as the interaction with unanalyzed libraries or with the environment (as in Fig. 1), and permits further abstractions (Sec. 4.6). Using $\widetilde{\mathrm{pre}}$ ensures that the target invariant T holds for all the (possibly infinite) sequences of choices made at each execution step, while pre would infer conditions for the invariant to hold for at least one sequence of choices, but not necessarily all (a laxer condition).

Blocking states. Any state σ without a successor satisfies $\forall X : \sigma \in \widetilde{\mathrm{pre}}(X)$, and so, $\sigma \in T \implies \sigma \in \mathrm{cond}(T)$. Such states correspond to a normal or abnormal program termination — e.g., the statement $y = 1/x$ generates a transition only from states where $x \neq 0$. In the following, we assume the absence of blocking states by adding transitions to self-looping states: error states transition to a self-loop $\omega \notin T$, and normal termination states transition to a self-loop $\alpha \in T$, so that no erroneous execution can stem from $\mathrm{cond}(T)$.

Approximation. Transition systems can become large or infinite, so that $\mathrm{inv}(I)$ and $\mathrm{cond}(T)$ cannot be computed efficiently or at all. We settle for sound approximations. Invariant sets are over-approximated in order to be certain

[3] In [4], Cousot notes it $sp(\tau*)$ and defines it rather as $\mathrm{lfp}\, \lambda X.I \cup \mathrm{post}(X)$. Both formulations are equivalent: both equal $\cup_{n \geq 0} \mathrm{post}^n(I)$ because post is a complete $\cup$−morphism in the complete lattice $(\mathcal{P}(\Sigma), \subseteq, \cup, \cap)$.

[4] In [2], Bourdoncle calls this set *always*(T) and defines it equivalently as $\mathrm{gfp}\, \lambda X.T \cap \widetilde{\mathrm{pre}}(X)$, but only considers the case where $\forall \sigma : |\mathrm{post}(\{\sigma\})| = 1$, i.e., $\widetilde{\mathrm{pre}} = \mathrm{pre}$.

to include all program behaviors. Sufficient condition sets are dually *under-approximated* as any subset $I' \subseteq \mathrm{cond}(T)$ still satisfies $\mathrm{inv}(I') \subseteq T$.

2.2 Applications of sufficient conditions

This section presents a few applications of computing an under-approximation of $\mathrm{cond}(T)$. The rest of the paper focuses on how to compute it effectively.

Given a set of initial states I, the subset of initial states that never lead to a run-time error nor assertion violation can be expressed as $I_{\overline{\omega}} \stackrel{\text{def}}{=} I \cap \mathrm{cond}(\Sigma \setminus \{\omega\})$. An analyzer computing an under-approximation of $I_{\overline{\omega}}$ infers sufficient initial conditions so that *all* executions are correct (i.e., never reach ω). Applied to a single function, it infers sufficient conditions on its parameters and global entry state ensuring its correct behavior. An application to software engineering is the automatic inference of method contracts. An application to optimizing compilation is run-time check hoisting: a fast version of the function without any check is called instead of the regular one if, at the function entry, sufficient conditions making these checks useless hold.

As last application, we consider the automatic inference of counter-examples. Given a set T of target states (e.g., erroneous states), we seek initial conditions such that *all* the executions *eventually* reach a state in T. In that case, Thm. 2.1 cannot be directly applied as it focuses on invariance properties while we are now interested in an inevitability property. We now show that this problem can nevertheless be reduced to an invariance one, of the form $\mathrm{cond}(T')$ for some T'. We use an idea proposed by Cousot et al. [5] for the analysis of termination (another inevitability property): we enrich the transition system with a counter variable l counting execution steps from some positive value down to 0 when reaching T. Given (Σ, τ) and $T \subseteq \Sigma$, we construct (Σ', τ') and T' as:

$$\Sigma' \stackrel{\text{def}}{=} \Sigma \times \mathbb{N}, \quad T' \stackrel{\text{def}}{=} \{\, (\sigma, l) \in \Sigma' \mid l > 0 \vee \sigma \in T \,\}$$

$$((\sigma, l), (\sigma', l')) \in \tau' \stackrel{\text{def}}{\iff} ((\sigma \notin T \wedge (\sigma, \sigma') \in \tau) \vee \sigma = \sigma' \in T) \wedge l = l' + 1$$

This transformation is always sound and sometimes complete:

Theorem 2.2 *If $(\sigma, l) \in \mathrm{cond}(T')$, then all the traces starting in σ eventually enter a state in T. If the non-determinism in τ is finite,*[5] *the converse holds.*

The restriction to finite non-determinism may hinder the analysis of fair systems, as an infinite number of countable choices must be performed, e.g.:

$$\texttt{while}\ ([0;1])\ \{\, n = [0; +\infty]; \texttt{while}\ (n > 0)\ \{\, n = n - 1 \,\} \,\}\ .$$[6]

[5] I.e., $\forall \sigma : \mathrm{post}(\{\sigma\})$ is finite, which is weaker than requiring a bounded non-determinism.
[6] Note that, if the number of infinite choices is bounded, they can be embedded as fresh non-initialized variables to obtain a program with finite non-determinism.

3 Backward Functions

Program semantics are not generally defined as monolithic transition systems, but rather as compositions of small reusable blocks. To each atomic language instruction i corresponds a forward transfer function post_i, and we will construct backward transfer functions directly from these post_i. Formally, given a function $f : \mathcal{P}(X) \to \mathcal{P}(Y)$, we define its backward version $\overleftarrow{f}$ as:

$$\overleftarrow{f} : \mathcal{P}(Y) \to \mathcal{P}(X) \text{ s.t. } \overleftarrow{f}(B) \stackrel{\text{def}}{=} \{ a \in X \mid f(\{a\}) \subseteq B \} \ . \tag{3}$$

We note immediately that $\overleftarrow{\text{post}} = \widetilde{\text{pre}}$. Moreover, $\overleftarrow{\cdot}$ enjoys useful properties. We list a few ones to give a gist of the underlying algebraic structure:

Theorem 3.1

(i) $\overleftarrow{f}$ *is a monotonic, complete* $\cap-$*morphism.*

(ii) $\overleftarrow{f}$ *is a sup-*$\cup-$*morphism:* $\cup_{i\in I} \overleftarrow{f}(B_i) \subseteq \overleftarrow{f}(\cup_{i\in I} B_i)$
(in general it is not a $\cup-$*morphism, nor is it strict, even when* f *is).*

(iii) *If* f *is a strict complete* $\cup-$*morphism, then* $A \subseteq \overleftarrow{f}(B) \iff f(A) \subseteq B$,
that is, we have a Galois connection: $\mathcal{P}(X) \underset{f}{\overset{\overleftarrow{f}}{\leftrightarrows}} \mathcal{P}(Y)$.

(iv) $\overleftarrow{f \cup g} = \overleftarrow{f} \cap \overleftarrow{g}$ *(note that* $\cup$, $\cap$, $\subseteq$ *are extended point-wise to functions).*

(v) $\overleftarrow{f \cap g} \supseteq \overleftarrow{f} \cup \overleftarrow{g}$ *(in general, the equality does not hold).*

(vi) *If* f *is monotonic, then* $\overleftarrow{f \circ g} \subseteq \overleftarrow{g} \circ \overleftarrow{f}$.

(vii) *If* f *is a strict complete* $\cup-$*morphism, then* $\overleftarrow{f \circ g} = \overleftarrow{g} \circ \overleftarrow{f}$.

(viii) $f \subseteq g \Longrightarrow \overleftarrow{g} \subseteq \overleftarrow{f}$.

(ix) *If* f *and* g *are strict complete* $\cup-$*morphisms, then* $f \subseteq g \iff \overleftarrow{g} \subseteq \overleftarrow{f}$.

(x) *If* f *is a strict complete* $\cup-$*morphism, then* $\overleftarrow{\lambda x.\text{lfp}_x\,(\lambda z.z \cup f(z))} = \lambda y.\text{gfp}_y(\lambda z.z \cap \overleftarrow{f}(z))$.

Property iv is useful to handle semantics expressed as systems of flow equations $\forall i : X_i = \cup_j F_{i,j}(X_j)$; we get: $\forall j : X_j = \cap_j \overleftarrow{F_{i,j}}(X_i)$. Compositional semantics make use of $\circ$, $\cup$, and nested least fixpoints (vii, iv, x). Properties iii and x generalize Thm. 2.1 and Eq. 2. Finally, viii–ix, turn forward over-approximations into backward under-approximations. All these properties will also be useful to design abstract operators for atomic statements, in Sec. 4.

4 Under-Approximated Polyhedral Operators

We use the results of the previous section to design practical backward operators sufficient to implement an analysis. We focus on numeric properties

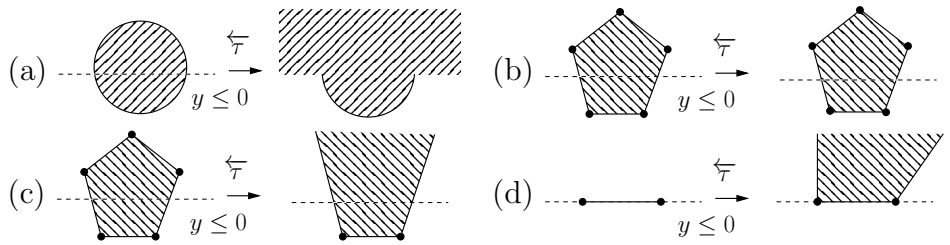


Fig. 2. Modeling the test $y \leq 0$ backwards in the concrete (a) and with polyhedra (b)–(d).

that we abstract using convex closed polyhedra (although the ideas we present can be used in other linear inequality domains, such as intervals or octagons). Familiarity with the over-approximating polyhedron domain [7] is assumed.

We assume a set $\mathcal{V}$ of variables with value in $\mathbb{Q}$. Environments $\rho \in \mathcal{E} \stackrel{\text{def}}{=} \mathcal{V} \to \mathbb{Q}$ map each variable to its value in $\mathbb{Q}$. A polyhedron P can be encoded as a set $C = \{ c_1, \ldots, c_n \}$ of affine constraints $c_i = (\boldsymbol{a_i} \cdot \boldsymbol{x} \geq b_i)$, which represents $\gamma_c(C) \stackrel{\text{def}}{=} \{ \rho \in \mathcal{E} \mid \forall (\boldsymbol{a} \cdot \boldsymbol{x} \geq b) \in C : \boldsymbol{a} \cdot \rho(\boldsymbol{x}) \geq b \}$, but also as a set of vertices and rays (V, R), so called generators, which represents $\gamma_g(V, R) \stackrel{\text{def}}{=} \{ \sum_{v \in V} \alpha_v v + \sum_{r \in R} \beta_r r \mid \alpha_v, \beta_r \geq 0, \sum_{v \in V} \alpha_v = 1 \}$. Here, $\boldsymbol{a}$ denotes a vector, $\cdot$ is the dot product, and $\rho(\boldsymbol{x})$ is the vector of variable values in environment ρ. Given a statement s, we denote by $\tau \llbracket s \rrbracket$ its forward concrete transfer function, and by $\overleftarrow{\tau} \llbracket s \rrbracket$ its backward version $\overleftarrow{\tau} \llbracket s \rrbracket \stackrel{\text{def}}{=} \overleftarrow{\tau \llbracket s \rrbracket}$.

Note that $\emptyset$ can always be used to under-approximate any $\overleftarrow{\tau} \llbracket s \rrbracket$, the same way over-approximating analyzers soundly bail-out with $\mathcal{E}$ in case of a time-out or unimplemented operation. Because backward operators are generally not strict (i.e., $\overleftarrow{f}(\emptyset) \neq \emptyset$, as the tests in Sec. 4.1), returning $\emptyset$ at some point does not prevent finding a non-empty sufficient condition at the entry point; it only means that the analysis forces some program branch to be dead.

4.1 Tests

We first consider simple affine tests $\boldsymbol{a} \cdot \boldsymbol{x} \geq b$. We have:

$$\tau \llbracket \boldsymbol{a} \cdot \boldsymbol{x} \geq b? \rrbracket R \stackrel{\text{def}}{=} \{ \rho \in R \mid \boldsymbol{a} \cdot \rho(\boldsymbol{x}) \geq b \}$$

$$\text{and so} \quad \overleftarrow{\tau} \llbracket \boldsymbol{a} \cdot \boldsymbol{x} \geq b? \rrbracket R = R \cup \{ \rho \in \mathcal{E} \mid \boldsymbol{a} \cdot \rho(\boldsymbol{x}) < b \}$$

On polyhedra, forward affine tests are handled exactly by simply adding the constraint. However, the result of a backward affine test on a closed convex set is generally not closed nor convex (see Fig. 2.a), so, we need an actual under-approximation. One solution is to remove $\boldsymbol{a} \cdot \boldsymbol{x} \geq b$ from the set C, as:

Theorem 4.1 $\gamma_c(C \setminus \{\boldsymbol{a} \cdot \boldsymbol{x} \geq b\}) \subseteq \overleftarrow{\tau} \llbracket \boldsymbol{a} \cdot \boldsymbol{x} \geq b? \rrbracket \gamma_c(C)$.

Sometimes, this results in the identity (Fig. 2.b) which is indeed a (trivial) under-approximation. More precise (i.e., *larger*) under-approximations can be computed by removing the constraints that are redundant in $C \cup \{\boldsymbol{a} \cdot \boldsymbol{x} \geq b\}$.

Intuitively, these are constraints that restrict $\gamma_c(C)$ in the half-space $\boldsymbol{a}\cdot\boldsymbol{x} < b$, while the test result is not restricted in this half-space (Fig. 2.c). In practice, we first add $\boldsymbol{a}\cdot\boldsymbol{x} \geq b$, then remove redundant constraints, then remove $\boldsymbol{a}\cdot\boldsymbol{x} \geq b$.

Consider now the degenerate case where $\gamma_c(C) \models \boldsymbol{a}\cdot\boldsymbol{x} = b$ (Fig. 2.d). Constraint representations are not unique, and different choices may result in different outcomes. To guide us, we exploit the fact that tests come in pairs, one for each program branch: while a forward semantics computes, at a branch split, $(Y, Z) = (\tau \{\!|\, \boldsymbol{a}\cdot\boldsymbol{x} \geq b? \,|\!\} X, \tau \{\!|\, \boldsymbol{a}\cdot\boldsymbol{x} < b? \,|\!\} X)$, the backward computation merges both branches as $X = \overleftarrow{\tau} \{\!|\, \boldsymbol{a}\cdot\boldsymbol{x} \geq b? \,|\!\} Y \cap \overleftarrow{\tau} \{\!|\, \boldsymbol{a}\cdot\boldsymbol{x} < b? \,|\!\} Z$. Assuming that $Y = \gamma_g(V_Y, R_Y)$ is degenerate, we construct a non-degenerate polyhedron before computing $\overleftarrow{\tau} \{\!|\, \boldsymbol{a}\cdot\boldsymbol{x} \geq b? \,|\!\}$ by adding the rays r from Z such that $\tau \{\!|\, \boldsymbol{a}\cdot\boldsymbol{x} \geq b? \,|\!\} \gamma_g(V_Y, R_Y \cup \{r\}) = \tau \{\!|\, \boldsymbol{a}\cdot\boldsymbol{x} \geq b? \,|\!\} \gamma_g(V_Y, R_Y)$. The effect is to create common rays in $\overleftarrow{\tau} \{\!|\, \boldsymbol{a}\cdot\boldsymbol{x} \geq b? \,|\!\} Y$ and $\overleftarrow{\tau} \{\!|\, \boldsymbol{a}\cdot\boldsymbol{x} < b? \,|\!\} Z$ to make the subsequent intersection as large as possible. This simple heuristic is sufficient to analyze Fig. 1 (where the degeneracy comes from the invariant $\mathtt{i} = 100$ at loop exit) but it is nevertheless fragile and begs to be improved.

To handle strict tests, we note that $\tau \{\!|\, \boldsymbol{a}\cdot\boldsymbol{x} \geq b? \,|\!\}$ over-approximates $\tau \{\!|\, \boldsymbol{a}\cdot\boldsymbol{x} > b? \,|\!\}$, and so, by Thm. 3.1.viii, $\overleftarrow{\tau} \{\!|\, \boldsymbol{a}\cdot\boldsymbol{x} > b? \,|\!\}$ can be under-approximated by $\overleftarrow{\tau} \{\!|\, \boldsymbol{a}\cdot\boldsymbol{x} \geq b? \,|\!\}$. Similarly for non-deterministic tests, $\tau \{\!|\, \boldsymbol{a}\cdot\boldsymbol{x} \geq [b;c]? \,|\!\} = \tau \{\!|\, \boldsymbol{a}\cdot\boldsymbol{x} \geq b? \,|\!\}$, and so, $\overleftarrow{\tau} \{\!|\, \boldsymbol{a}\cdot\boldsymbol{x} \geq [b;c]? \,|\!\}$ is modeled as $\overleftarrow{\tau} \{\!|\, \boldsymbol{a}\cdot\boldsymbol{x} \geq b? \,|\!\}$. We will see in Sec. 4.6 that non-linear tests can be abstracted into such non-deterministic affine ones. Finally, boolean combinations of tests are handled as follows, using Thm. 3.1.iv,vii: [7]

$$\tau \{\!|\, t_1 \vee t_2 \,|\!\} = \tau \{\!|\, t_1 \,|\!\} \cup \tau \{\!|\, t_2 \,|\!\} \text{ and so } \overleftarrow{\tau} \{\!|\, t_1 \vee t_2 \,|\!\} = \overleftarrow{\tau} \{\!|\, t_1 \,|\!\} \cap \overleftarrow{\tau} \{\!|\, t_2 \,|\!\}$$

$$\tau \{\!|\, t_1 \wedge t_2 \,|\!\} = \tau \{\!|\, t_2 \,|\!\} \circ \tau \{\!|\, t_1 \,|\!\} \text{ and so } \overleftarrow{\tau} \{\!|\, t_1 \wedge t_2 \,|\!\} = \overleftarrow{\tau} \{\!|\, t_1 \,|\!\} \circ \overleftarrow{\tau} \{\!|\, t_2 \,|\!\}$$

For instance, $\overleftarrow{\tau} \{\!|\, \boldsymbol{a}\cdot\boldsymbol{x} = [b;c]? \,|\!\} = \overleftarrow{\tau} \{\!|\, \boldsymbol{a}\cdot\boldsymbol{x} \geq b? \,|\!\} \circ \overleftarrow{\tau} \{\!|\, (-\boldsymbol{a})\cdot\boldsymbol{x} \geq -c? \,|\!\}$.

4.2 Projection

Given a variable V, projecting it forgets its value:

$$\tau \{\!|\, V := ? \,|\!\} R \stackrel{\text{def}}{=} \{\, \rho[V \mapsto v] \mid \rho \in R, v \in \mathbb{Q} \,\}$$

$$\text{and so} \quad \overleftarrow{\tau} \{\!|\, V := ? \,|\!\} R = \{\, \rho \in \mathcal{E} \mid \forall v \in \mathbb{Q} : \rho[V \mapsto v] \in R \,\}$$

We have the following property:

Theorem 4.2 *If R is convex closed, then $\overleftarrow{\tau} \{\!|\, V := ? \,|\!\} R$ is either R or $\emptyset$.*

The projection can be efficiently and exactly implemented for polyhedra as: if $\tau \{\!|\, V := ? \,|\!\} P = P$ then $\overleftarrow{\tau} \{\!|\, V := ? \,|\!\} P = P$, otherwise $\overleftarrow{\tau} \{\!|\, V := ? \,|\!\} P = \emptyset$. Adding and removing an uninitialized variable can then be derived as follows:

[7] We avoid the use of $\cap$ for $\wedge$ as it does not behave well with respect to $\overleftarrow{\cdot}$, see Thm. 3.1.v.

$$\overleftarrow{\tau}\{\!|\,\texttt{del}\ V\,|\!\} = \tau\{\!|\,\texttt{add}\ V\,|\!\}$$
$$\overleftarrow{\tau}\{\!|\,\texttt{add}\ V\,|\!\} = \tau\{\!|\,\texttt{del}\ V\,|\!\} \circ \overleftarrow{\tau}\{\!|\,V := ?\,|\!\}$$

4.3 Assignments

By Thm. 3.1.viii, and given that the forward projection over-approximates any assignment, the backward projection can be used to under-approximate any assignment, but this is rather coarse. More interestingly, general assignments can be reduced to tests by introducing a temporary variable V'. We note $[V'/V]$ the renaming of V as V'. We have:

$$\tau\{\!|\,V := e\,|\!\} = [V/V'] \circ \tau\{\!|\,\texttt{del}\ V\,|\!\} \circ \tau\{\!|\,V' = e?\,|\!\} \circ \tau\{\!|\,\texttt{add}\ V'\,|\!\}$$
$$\text{and so } \overleftarrow{\tau}\{\!|\,V := e\,|\!\} = \overleftarrow{\tau}\{\!|\,\texttt{add}\ V'\,|\!\} \circ \overleftarrow{\tau}\{\!|\,V' = e?\,|\!\} \circ \overleftarrow{\tau}\{\!|\,\texttt{del}\ V\,|\!\} \circ [V'/V]$$

In case of degeneracy on a test argument, Sec. 4.1 relied on rays provided by another polyhedron to guide the operation. We do not have another polyhedron here, but we know that the test is followed by a projection (as part of $\overleftarrow{\tau}\{\!|\,\texttt{add}\ V'\,|\!\}$), hence, the heuristic is modified to use the rays V' and $-V'$. Intuitively, we try to maximize the set of environments ρ such that the result of the test contains $\{\,\rho[V' \mapsto v] \mid v \in \mathbb{Q}\,\}$, and so, will be kept by $\overleftarrow{\tau}\{\!|\,V' := ?\,|\!\}$.

Moreover, some restricted yet useful classes of assignments enjoy more direct abstractions, based solely on forward operators, such as:

Theorem 4.3

(i) $\overleftarrow{\tau}\{\!|\,V := [a;b]\,|\!\} = \tau\{\!|\,V := ?\,|\!\} \circ (\tau\{\!|\,V := V - a\,|\!\} \cap \tau\{\!|\,V := V - b\,|\!\}) \circ \tau\{\!|\,V \geq a? \wedge V \leq b?\,|\!\})$.

(ii) $\overleftarrow{\tau}\{\!|\,V := V + [a;b]\,|\!\} = \tau\{\!|\,V := V - a\,|\!\} \cap \tau\{\!|\,V := V - b\,|\!\}$.

(iii) $\overleftarrow{\tau}\{\!|\,V := W\,|\!\} = \tau\{\!|\,V := ?\,|\!\} \circ \tau\{\!|\,V = W?\,|\!\}$ *(when $V \neq W$).*

(iv) *If $V := e$ is invertible, i.e., there exists an expression e^{-1} such that $\tau\{\!|\,V := e^{-1}\,|\!\} \circ \tau\{\!|\,V := e\,|\!\} = \tau\{\!|\,V := e\,|\!\} \circ \tau\{\!|\,V := e^{-1}\,|\!\} = \lambda R.R$, then $\overleftarrow{\tau}\{\!|\,V := e\,|\!\} = \tau\{\!|\,V := e^{-1}\,|\!\}$ — e.g., $V := \sum_W \alpha_W W$ with $\alpha_V \neq 0$.*

4.4 Lower widening

Invariance semantics by abstract interpretation feature least fixpoints, e.g., to handle loops and solve equation systems. Traditionally, they are solved by iteration with an upper convergence acceleration operator, the widening ∇ [4]. To compute sufficient conditions, we under-approximate greatest fixpoints instead (Eq. 2 and Thm. 3.1.x). We thus define a lower widening $\underline{\nabla}$ obeying:

(i) $\gamma(A \mathbin{\underline{\nabla}} B) \subseteq \gamma(A) \cap \gamma(B)$.

(ii) For any sequence $(X_n)_{n\in\mathbb{N}}$, the sequence $Y_0 = X_0$, $Y_{n+1} = Y_n \mathbin{\underline{\nabla}} X_{n+1}$

stabilizes: $\exists i : Y_{i+1} = Y_i$.

As a consequence, for any under-approximation $F^\sharp$ of a concrete operator F, and any X_0, the sequence $X_{i+1} = X_i \underline{\triangledown} F^\sharp(X_i)$ stabilizes in finite time to some X_δ; moreover, this X_δ satisfies $\gamma(X_\delta) \subseteq \mathrm{gfp}_{\gamma(X_0)} F$ [4].

On polyhedra, by analogy with the widening $\triangledown$ [7] that keeps only stable constraints, we define a lower widening $\underline{\triangledown}$ that keeps only stable generators. Let V_P and R_P denote the vertices and rays of a polyhedron $P = \gamma_g(V_P, R_P)$. We define $\underline{\triangledown}$ formally as:

$$\begin{aligned} V_{A \underline{\triangledown} B} &\stackrel{\text{def}}{=} \{\, v \in V_A \mid v \in B \,\} \\ R_{A \underline{\triangledown} B} &\stackrel{\text{def}}{=} \{\, r \in R_A \mid B \oplus \mathbb{R}^+ r = B \,\} \end{aligned} \tag{4}$$

where $\oplus$ denotes the Minkowski sum ($A \oplus B \stackrel{\text{def}}{=} \{\, a + b \mid a \in A,\, b \in B \,\}$) and $\mathbb{R}^+ r$ denotes the set $\{\, \lambda r \mid \lambda \geq 0 \,\}$). We have:

Theorem 4.4 *$\underline{\triangledown}$ is a lower widening.*

Generator representations are not unique, and the output of $\underline{\triangledown}$ depends on the choice of representation. The same issue occurs for the standard widening. We can use a similar fix: we add to $A \underline{\triangledown} B$ any generator from B that is redundant with a generator in A. Our lower widening can also be refined in a classic way by permitting thresholds: given a finite set of vertices (resp. rays), each vertex v (resp. ray r) included in both polyhedra A and B ($v \in A \wedge v \in B$, resp. $A \oplus \mathbb{R}^+ r = A \wedge B \oplus \mathbb{R}^+ r = B$) is added to $A \underline{\triangledown} B$. As for any extrapolation operator, the effectiveness of $\underline{\triangledown}$ will need, in future work, to be assessed in practice. There is ample room for improvement and adaptation.

Lower widenings are introduced in [4] but, up to our knowledge, and unlike (upper) widenings, no practical instance on infinite domains has ever been designed. Lower widenings are designed to "jump below" fixpoints (hence performing an induction) and should not be confused with narrowing operators that "stay above" fixpoints (performing a refinement).

4.5 Joins

In invariance analyses, unions of environment sets are computed at every control flow join. Naturally, a large effort in abstract analysis design is spent designing precise and efficient over-approximations of unions. By the duality of Thm. 3.1.iv, such joins do not occur in sufficient condition analyses; they are replaced with intersections $\cap$ at control-flow splits, and these are easier to abstract in most domain (e.g., polyhedra). Hence, we avoid the issue of designing under-approximations of *arbitrary* unions. We do under-approximate unions as part of test operators (Sec. 4.1), but these have a very specific form which helped us design the approximation.

4.6 Expression approximation

We focused previously on affine tests and assignments because they match the expressive power of polyhedra, but programs feature more complex expressions. In [13], we proposed to solve this problem for over-approximating transfer functions using an expression abstraction mechanism. We noted $e \sqsubseteq_D f$ the fact that f approximates e on D, i.e., $\forall \rho \in D : [\![e]\!]\rho \subseteq [\![f]\!]\rho$, where $[\![\cdot]\!] : \mathcal{E} \to \mathcal{P}(\mathbb{Q})$ evaluates an expression in an environment. Then:

$$\text{if } R \subseteq D \text{ then } \tau \,\{\!|\, V := e \,|\!\}\, R \subseteq \tau \,\{\!|\, V := f \,|\!\}\, R \text{ and } \tau \,\{\!|\, e? \,|\!\}\, R \subseteq \tau \,\{\!|\, f? \,|\!\}\, R$$

so, in the abstract, e can be replaced with f if the argument $A^\sharp$ satisfies $e \sqsubseteq_{\gamma(A^\sharp)} f$. We now show that this method also works for under-approximations:

Theorem 4.5 *If $e \sqsubseteq_D f$, we have:*

(i) $\overleftarrow{\tau} \,\{\!|\, V := e \,|\!\}\, R \supseteq (\overleftarrow{\tau} \,\{\!|\, V := f \,|\!\}\, R) \cap D$.

(ii) $\overleftarrow{\tau} \,\{\!|\, e? \,|\!\}\, R \supseteq (\overleftarrow{\tau} \,\{\!|\, f? \,|\!\}\, R) \cap D$.

We study the case of abstract assignments (tests are similar): $\overleftarrow{\tau} \,\{\!|\, V := e \,|\!\}^\sharp\, A^\sharp$ can be replaced with $\overleftarrow{\tau} \,\{\!|\, V := f \,|\!\}^\sharp\, A^\sharp \cap^\sharp D^\sharp$ if $e \sqsubseteq_{\gamma(D^\sharp)} f$. One way to construct f is to use the "linearization" from [13]: it converts an arbitrary expression into an expression of the form $\sum_V \alpha_V V + [a; b]$ by performing interval arithmetics on non-linear parts, using variable bounds from $D^\sharp$. The theorem does not make any hypothesis on the choice of D (unlike the case of forward analysis). A smaller D improves the precision of f by making $[a; b]$ tighter, but, as we want to maximize the result of the backward assignment, we should avoid discarding states in $\overleftarrow{\tau} \,\{\!|\, V := f \,|\!\} R$ but not in $\overleftarrow{\tau} \,\{\!|\, V := f \,|\!\} R \cap D$. In practice, we use for D the result $\gamma(D^\sharp)$ of a prior invariance analysis as we know that, in the concrete, $\overleftarrow{\tau} \,\{\!|\, V := e \,|\!\}\, R \subseteq \gamma(D^\sharp)$. For instance, the assignment $\overleftarrow{\tau} \,\{\!|\, X \leftarrow Y \times Z \,|\!\}^\sharp\, R^\sharp$ will be replaced with $\overleftarrow{\tau} \,\{\!|\, X \leftarrow Y \times [0; 1] \,|\!\}^\sharp\, R^\sharp \cap^\sharp D^\sharp$ if the invariant $\gamma(D^\sharp)$ before the assignment implies that $Z \in [0; 1]$.

It may seem counter-intuitive that *over-approximating* expressions results in *under-approximating* backward transfer functions. Observe that over-approximations enlarge the non-determinism of expressions, and so, make it less likely to find sufficient conditions holding for all cases.

4.7 Implementation

We have implemented a proof-of-concept analyzer [14] that infers sufficient pre-conditions for programs written in a toy language to never violate any user-specified assertion. It first performs a classic forward over-approximating analysis, followed with a backward under-approximating one. All the abstract operators are implemented with polyhedra, on top of the Apron library [10]. It is able to find the sufficient condition $\mathtt{j} \in [0; 5]$ in the example of Fig. 1. We also analyzed the BubbleSort example that introduced polyhedral analysis [7].

5 Related Work

Since their introduction by Dijkstra [8], weakest (liberal) preconditions have been much studied, using a variety of inference and checking methods, including interactive theorem proving [9] and automatic finite-state computations. These methods are *exact* (possibly with respect to an abstract model over-approximating the concrete system, so that sufficient conditions on the model do not always give sufficient conditions for the original system). Fully automatic methods based on under-approximations are less common.

Bourdoncle introduces [2] sufficient conditions, denoted $\mathit{always}(T)$, but only focuses on deterministic systems (i.e., $\widetilde{\mathrm{pre}} = \mathrm{pre}$). He also mentions that classic domains, such as intervals, are inadequate to express under-approximations as they are not closed under complementation, but he does not propose an alternative. Moy [15] solves this issue by allowing disjunctions of abstract states (they correspond to path enumerations and can grow arbitrarily large). Lev-Ami et al. [11] derive under-approximations from over-approximations by assuming, similarly, that abstract domains are closed by complementation (or negation, when seen as formulas). Brauer et al. [3] employ boolean formulas on a bit-vector (finite) domain. These domains are more costly than classic convex ones, and our method is not limited to them.

Schmidt [17] defines Galois Connections (and so best operators) for all four backward/forward over-/under-approximation cases using a higher-order powerset construction. Massé [12] proposes an analysis parametrized by arbitrary temporal properties, including $\widetilde{\mathrm{pre}}$ operators, based on abstract domains for lower closure operators. We shy away from higher-order constructions. We may lose optimality and generality, but achieve a more straightforward and, we believe, practical framework. In particular, we do not change the semantics of abstract elements, but only add new transfer functions, and achieve the same algorithmic complexity as forward analyses.

Cousot et al. [6] propose a backward precondition analysis for contracts. It differs from the weakest precondition approach we follow in its treatment of non-determinism: it keeps states that, for some sequence of choices (but not necessarily all), give rise to a non-erroneous execution. Our handling of inevitability is directly inspired from Cousot et al. [5].

6 Conclusion

In this article, we have discussed the inference of sufficient conditions by abstract interpretation. We have presented general properties of backward under-approximated semantics, and proposed example transfer functions in the polyhedra domain. Much work remains to be done, including designing new under-approximated operators (tests and lower widenings, in particular),

considering new domains, experimenting on realistic programs. Our construction and results are very preliminary and remain mostly untried; our hope is only to convince the reader that this constitutes a fruitful avenue of research.

References

[1] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis and verification of aerospace software by abstract interpretation. In *AIAA Infotech@Aerospace*, number 2010-3385, pages 1–38. AIAA, Apr. 2010.

[2] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Proc. of the ACM Conf. on Prog. Lang. Design and Implementation (PLDI'93)*, pages 46–55. ACM, Jun. 1993.

[3] J. Brauer and A. Simon. Inferring definite counterexamples through under-approximation. In *NASA Formal Methods*, volume 7226 of *LNCS*, Apr. 2012.

[4] P. Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes (in French)*. Thèse d'État ès sciences mathématiques, Université scientifique et médicale de Grenoble, Grenoble, France, 21 Mar. 1978.

[5] P. Cousot and R. Cousot. An abstract interpretation framework for termination. In *Conference Record of the 39$^{\text{th}}$ Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 245–258, Philadelphia, PA, January 25-27 2012. ACM Press, New York.

[6] P. Cousot, R. Cousot, and F. Logozzo. Precondition inference from intermittent assertions and application to contracts on collections. In *Proc. of the 12th Int. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI;11)*, volume 6538 of *LNCS*, pages 150–168. Springer, Jan. 2011.

[7] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. of the 5th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'78)*, pages 84–97. ACM Press, 1978.

[8] E. W. Dijkstra. Guarded commands, non-determinacy and formal derivation of programs. *Comm. ACM*, 18(8):453–457, 1975.

[9] C. Flanagan, R. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proc. of the SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'02)*, pages 234–245. ACM, June 2002.

[10] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *Proc. of the 21th Int. Conf. on Computer Aided Verification (CAV'09)*, volume 5643 of *LNCS*, pages 661–667. Springer, June 2009.

[11] T. Lev-Ami, M. Sagiv, T. Reps, and S. Gulwani. Backward analysis for inferring quantified pre-conditions. Technical Report TR-2007-12-01, Tel Aviv University, Dec. 2007.

[12] D. Massé. *Temporal Property-driven Verification by Abstract Interpretation*. PhD thesis, École Polytechnique, Palaiseau, France, Dec. 2002.

[13] A. Miné. Symbolic methods to enhance the precision of numerical abstract domains. In *Proc. of the 7th Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI'06)*, volume 3855 of *LNCS*, pages 348–363. Springer, Jan. 2006.

[14] A. Miné. The Banal static analyzer prototype, 2012. http://www.di.ens.fr/~mine/banal.

[15] Y. Moy. Sufficient preconditions for modular assertion checking. In *Proc. of the 9th Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI'08)*, volume 4905 of *LNCS*, pages 188–202. Springer, Jan 2008.

[16] X. Rival. Understanding the origin of alarms in Astrée. In *Proc. of the 12th Int. Symp. on Static Analysis (SAS'05)*, volume 3672 of *LNCS*, pages 303–319. Springer, Sep. 2005.

[17] D. A. Schmidt. Closed and logical relations for over- and under-approximation of powersets. In *Proc. of the 11th Int. Symp. on Static Analysis (SAS'04)*, volume 3148, pages 22–37. Springer, Aug. 2004.

A Example Analyses

The analysis method described in the paper has been implemented in a proof-of-concept prototype analyzer. It analyzes a small toy language with a C-inspired syntax and numeric data-types. We present some results obtained with this analyzer on toy examples. The source code of the analyzer, more examples, as well as the ability to run custom analyses on our server are available through a web-based analysis interface at http://www.di.ens.fr/~mine/banal.

A.1 Simple loop

We start with the simple loop example from Fig. 1. In our language syntax, the example is written as:

```
integer j = [0;10], i = 0;
void main() {
  (6) while (7) (i < 100) {
    (8) i = i + 1;
    (9) j = j + [0;1];
    (10)
  }
  (12) assert (j <= 105);
  (14)
}
```

where each (i) denotes a program point.

We first perform a standard forward over-approximating analysis with polyhedra, and iterate the loop body until a fixpoint is reached. To ensure an efficient analysis, we use a widening at point 7, just before performing the loop test when entering the loop for the first time or after a loop iteration. To improve the precision, we use two known technique: we replace the first widening applications with a plain join (known as widening "with delay"), and apply one decreasing iteration after the increasing iteration with widening has stabilized (a simple form of narrowing). We show below the loop iterates (7–10) and the analysis after the loop (12, 14):

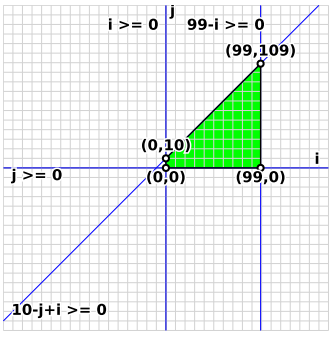
	up iter 1	up iter 2 ($\cup$)	up iter 3 ($\cup$)	up iter 4 (∇)
(7)	$\mathtt{i}=0\wedge-\mathtt{j}\geq-10\wedge\mathtt{j}\geq0$	$-\mathtt{i}\geq-1\wedge\mathtt{j}\geq0\wedge\mathtt{i}-\mathtt{j}\geq-10\wedge\mathtt{i}\geq0$	$-\mathtt{i}\geq-2\wedge\mathtt{j}\geq0\wedge\mathtt{i}-\mathtt{j}\geq-10\wedge\mathtt{i}\geq0$	$\mathtt{j}\geq0\wedge\mathtt{i}-\mathtt{j}\geq-10\wedge\mathtt{i}\geq0$
(8)	$\mathtt{i}=0\wedge-\mathtt{j}\geq-10\wedge\mathtt{j}\geq0$	$-\mathtt{i}\geq-1\wedge\mathtt{j}\geq0\wedge\mathtt{i}-\mathtt{j}\geq-10\wedge\mathtt{i}\geq0$	$-\mathtt{i}\geq-99\wedge\mathtt{j}\geq0\wedge\mathtt{i}-\mathtt{j}\geq-10\wedge\mathtt{i}\geq0$	$-\mathtt{i}\geq-99\wedge\mathtt{j}\geq0\wedge\mathtt{i}-\mathtt{j}\geq-10\wedge\mathtt{i}\geq0$
(9)	$\mathtt{i}=1\wedge-\mathtt{j}\geq-10\wedge\mathtt{j}\geq0$	$-\mathtt{i}\geq-2\wedge\mathtt{j}\geq0\wedge\mathtt{i}-\mathtt{j}\geq-9\wedge\mathtt{i}\geq1$	$-\mathtt{i}\geq-100\wedge\mathtt{j}\geq0\wedge\mathtt{i}-\mathtt{j}\geq-9\wedge\mathtt{i}\geq1$	$-\mathtt{i}\geq-100\wedge\mathtt{j}\geq0\wedge\mathtt{i}-\mathtt{j}\geq-9\wedge\mathtt{i}\geq1$
(10)	$\mathtt{i}=1\wedge-\mathtt{j}\geq-11\wedge\mathtt{j}\geq0$	$-\mathtt{i}\geq-2\wedge\mathtt{j}\geq0\wedge\mathtt{i}-\mathtt{j}\geq-10\wedge\mathtt{i}\geq1$	$-\mathtt{i}\geq-100\wedge\mathtt{j}\geq0\wedge\mathtt{i}-\mathtt{j}\geq-10\wedge\mathtt{i}\geq1$	$-\mathtt{i}\geq-100\wedge\mathtt{j}\geq0\wedge\mathtt{i}-\mathtt{j}\geq-10\wedge\mathtt{i}\geq1$

	down iter 1
(7)	$-\mathtt{i} \geq -100 \wedge \mathtt{j} \geq 0 \wedge \mathtt{i}-\mathtt{j} \geq -10 \wedge \mathtt{i} \geq 0$
(8)	$-\mathtt{i} \geq -99 \wedge \mathtt{j} \geq 0 \wedge \mathtt{i}-\mathtt{j} \geq -10 \wedge \mathtt{i} \geq 0$
(9)	$-\mathtt{i} \geq -100 \wedge \mathtt{j} \geq 0 \wedge \mathtt{i}-\mathtt{j} \geq -9 \wedge \mathtt{i} \geq 1$
(10)	$-\mathtt{i} \geq -100 \wedge \mathtt{j} \geq 0 \wedge \mathtt{i}-\mathtt{j} \geq -10 \wedge \mathtt{i} \geq 1$
(12)	$\mathtt{i} = 100 \wedge -\mathtt{j} \geq -110 \wedge \mathtt{j} \geq 0$
(14)	$\mathtt{i} = 100 \wedge -\mathtt{j} \geq -105 \wedge \mathtt{j} \geq 0$

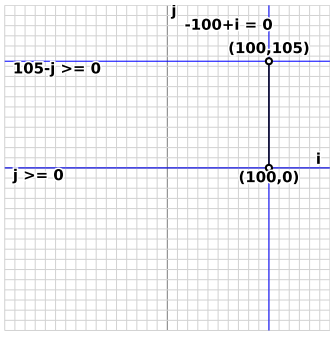
We now present the backward under-approximating analysis. We also use a narrowing "with delay" by replacing the first lower widening at 7 with a plain meet.

	down iter 1 ($\cap$)	down iter 2 ($\underline{\triangledown}$)
(14)	$\mathtt{i} = 100 \wedge -\mathtt{j} \geq -105 \wedge \mathtt{j} \geq 0$	
(12)	$\mathtt{i} = 100 \wedge -\mathtt{j} \geq -105 \wedge \mathtt{j} \geq 0$	
(10)	$-\mathtt{i} \geq -100 \wedge \mathtt{j} \geq 0 \wedge \mathtt{i}-\mathtt{j} \geq -10 \wedge \mathtt{i} \geq 1$	$-\mathtt{i} \geq -100 \wedge \mathtt{j} \geq 0 \wedge \mathtt{i}-\mathtt{j} \geq -5 \wedge \mathtt{i} \geq 1$
(9)	$-\mathtt{i} \geq -100 \wedge \mathtt{j} \geq 0 \wedge \mathtt{i}-\mathtt{j} \geq -9 \wedge \mathtt{i} \geq 1$	$-\mathtt{i} \geq -100 \wedge \mathtt{j} \geq 0 \wedge \mathtt{i}-\mathtt{j} \geq -4 \wedge \mathtt{i} \geq 1$
(8)	$-\mathtt{i} \geq -99 \wedge \mathtt{j} \geq 0 \wedge \mathtt{i}-\mathtt{j} \geq -10 \wedge \mathtt{i} \geq 0$	$-\mathtt{i} \geq -99 \wedge \mathtt{j} \geq 0 \wedge \mathtt{i}-\mathtt{j} \geq -5 \wedge \mathtt{i} \geq 0$
(7)	$-\mathtt{i} \geq -100 \wedge \mathtt{j} \geq 0 \wedge \mathtt{i}-\mathtt{j} \geq -5 \wedge \mathtt{i} \geq 0$	$-\mathtt{i} \geq -100 \wedge \mathtt{j} \geq 0 \wedge \mathtt{i}-\mathtt{j} \geq -5 \wedge \mathtt{i} \geq 0$
(6)		$\mathtt{i} = 0 \wedge -\mathtt{j} \geq -5 \wedge \mathtt{j} \geq 0$

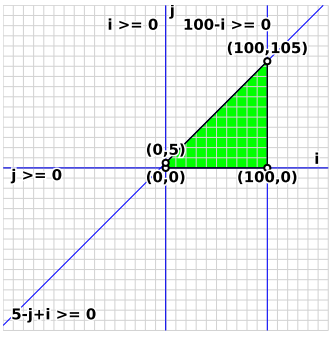
The most interesting part is the backward analysis of the loop test `i < 100`. A necessary loop condition at 7 is synthesized from the two conditions at 8 and 12, by applying respectively $\overleftarrow{\tau}\{\!|\, \mathtt{i} \leq 99 \,|\!\}$ and $\overleftarrow{\tau}\{\!|\, \mathtt{i} \geq 100 \,|\!\}$. This is illustrated graphically below:



(8)



(12)



(7)

A.2 Bubble sort

We now consider the Bubble sort example that illustrated the seminal work of Cousot and Halbwachs on polyhedral analysis [7]. Adapted to our toy language, this gives:

```
integer N, B, J, T;
void main() {
  (10)  B = N;
  (11)  while (12)  (B >= 1) {
    (13)  J = 1;
    (14)  T = 0;
    (15)  while (16)  (J <= B - 1) {
      (17)  assert (J >= 1 && J <= N && J + 1 >= 1 && J + 1 <= N);
      (18)  if ([0;1] == 1) { (19)  T = J; (20)  } else { (21)  };
      (22)  J = J + 1;
    (23)  }
  (25)  if (T == 0) { (26)  return; (27)  } else { (28)  };
  (29)  B = T;
  (30)  }
(33)  }
```

Note that the array accesses at indices `J` and `J+1` at 17 have been replaced with array bound check assertions, and the comparison of two array elements at 18 has been replaced with a non-deterministic test `[0;1]==1`.

We first analyze the correct version above. All our analyses unroll both loops twice, delay the widening and the lower widening twice, and use two decreasing iterations to refine the forward analysis. The forward analysis infers that, at the end of the program, $\mathtt{B} \leq \mathtt{N}$. The backward analysis infers no relation at the beginning of the program ($\top$), i.e., the program never performs any array bound error, which is the expected result.

We then analyze an incorrect version, where `B` is initialized to `N+1` instead of `N` at 10:

```
  (10)  B = N + 1;
```

The forward analysis infers no bound on `N` at the end of the program. However, the backward analysis finds that $\mathtt{N} \leq 0$ is a sufficient condition at the beginning of the program for the program to never performs a bound check error. The result is expected: obviously, the condition is sufficient to prevent the program to enter the inner loop (however, the program can enter the outer loop as, for $\mathtt{N} = 0$, $\mathtt{B} = \mathtt{N} + 1 = 1$); while, when $\mathtt{N} \geq 1$, the program necessarily enters the inner loop and performs a bound check error when `J` reaches $\mathtt{B} - 1 = \mathtt{N}$.

Finally, we analyze the same incorrect version where the assertion is changed to *force* an array bound check error:

```
      (17)  assert (!(J >= 1 && J <= N && J + 1 >= 1 && J + 1 <= N));
```

In this case, our analysis infers that $\mathtt{N} = 1$ is a sufficient initial condition so that, every time program point 17 is reached, an array bound check failure occurs. In fact, this provides a counter-example which ensures that the program necessarily fails (however, our analysis does not currently infers that program point 17 is necessarily reached).

B Proofs

B.1 Proof of Thm. 2.1

Proof. We prove: (1) $\forall T, X : \mathrm{inv}(\mathrm{cond}(T)) \subseteq T$ and (2) $\mathrm{inv}(X) \subseteq T \Longrightarrow X \subseteq \mathrm{cond}(T)$.

This is actually a special case and a consequence of the properties from Thm. 3.1, that are proved below. Indeed, by definition, $\mathrm{inv}(I) = \mathrm{lfp}_I\, \lambda X.X \cup \mathrm{post}(X)$. By Thm. 3.1.x, $\overleftarrow{\mathrm{inv}}(T) = \mathrm{gfp}_T\, \lambda X.X \cap \overleftarrow{\mathrm{post}}(X)$. We will also prove with Thm. 3.1 that $\overleftarrow{\mathrm{post}} = \widetilde{\mathrm{pre}}$, so that $\overleftarrow{\mathrm{inv}}(T) = \mathrm{gfp}_T\, \lambda X.X \cap \widetilde{\mathrm{pre}}(X) = \mathrm{cond}(T)$. We will also prove as a side-effect of Thm. 3.1.x that inv is a strict complete $\cup$−morphism. By Thm. 3.1.iii, $(\mathrm{inv}, \mathrm{cond})$ then forms a Galois connection and we have $\mathrm{inv}(X) \subseteq T \iff X \subseteq \mathrm{cond}(T)$. This implies immediately (2). By setting $X = \mathrm{cond}(T)$, this also implies (1). □

B.2 Proof of Thm. 2.2

Proof.

(i) We prove: If $(\sigma_0, l) \in \mathrm{cond}(T')$, then all the traces starting in σ_0 eventually enter a state in T.
Let us note $S' \stackrel{\text{def}}{=} \mathrm{cond}(T')$ in the transition system (Σ', τ'). Consider a state $(\sigma_0, l) \in S'$ and a maximal trace $t = \sigma_0, \ldots, \sigma_i, \ldots$ for (Σ, τ) starting in state σ_0. As we assumed that (Σ, τ) has no blocking state, the maximal trace t is infinite. We note $m \stackrel{\text{def}}{=} \min(l, \max\{\, i \mid \forall j < i : \sigma_j \notin T \,\})$. As $m \leq l$, m is finite. We construct the trace t' in (Σ', τ') as follows: $t' \stackrel{\text{def}}{=} (\sigma_0, l), (\sigma_1, l-1), \ldots, (\sigma_m, l-m)$. Then t' obeys τ'. As moreover, $(\sigma_0, l) \in S' = \mathrm{cond}(T')$, we have $\forall i : (\sigma_i, l_i) \in T'$. If $m = l$, this gives $l - m = 0$, and so $\sigma_m \in T$ by definition of T'. If $m < l$, then $m = \max\{\, i \mid \forall j < i : \sigma_j \notin T \,\})$, which implies $\sigma_m \in T$. We note that all the traces starting in state σ_0 reach a state in T in l steps or less.

(ii) We prove: If the non-determinism in τ is finite, and all the traces starting in σ eventually enter state T, then $\exists l : (\sigma, l) \in \mathrm{cond}(T')$.
Consider a state $\sigma \in \Sigma$ such that all the traces in (Σ, τ) from σ eventually reach a state in T. For each maximal such trace, we consider its finite prefix until it reaches T, i.e., $t = \sigma_1, \ldots, \sigma_{|t|}$ such that $\sigma_1 = \sigma$, $\sigma_{|t|} \in T$,

and $i < |t| \implies \sigma_i \notin T$. The set of all these finite trace prefixes forms a tree rooted at σ. By hypothesis, this tree has no infinite path. We now use the extra hypothesis that the non-determinism in τ is finite, which ensures that the tree is finitely branching. By the contrapositive of König's lemma, the tree is then finite. We denote by l its (finite) depth.

We now argue that $(\sigma, l) \in S' \stackrel{\text{def}}{=} \text{cond}(T')$. It is sufficient to prove that, for each finite trace $t' \stackrel{\text{def}}{=} (\sigma_0, l_0), \ldots, (\sigma_n, l_n)$ for (Σ', τ') starting in state $(\sigma_0, l_0) \stackrel{\text{def}}{=} (\sigma, l)$, we have $\forall i : (\sigma_i, l_i) \in T'$. By definition of τ', $l_i = l - i$. Thus, if $n < l$, then $\forall i : l_i > 0$ and the property is obvious. Otherwise, $n \geq l$, and it is sufficient to prove that $\exists i : \sigma_i \in T$. Suppose that this is not the case and $\forall i : \sigma_i \notin T$. Then, by definition of τ', the trace $t = \sigma_0, \ldots, \sigma_l$ in Σ obeys τ. We have constructed a trace for τ starting in σ with length strictly greater than l that does not encounter a state in T, which contradicts the definition of l.

□

B.3 Proof of Thm. 3.1

Proof.

(i) We prove: $\overleftarrow{f}$ is a monotonic, complete $\cap$−morphism.
If $A \subseteq B$, then $f(\{a\}) \subseteq A$ implies $f(\{a\}) \subseteq B$, and so, $\overleftarrow{f}(A) \subseteq \overleftarrow{f}(B)$, which proves the monotony. Moreover, $\overleftarrow{f}(\cap_{i\in I} B_i) = \{\, a \mid f(\{a\}) \subseteq \cap_{i\in I} B_i \,\} = \{\, a \mid \wedge_{i\in I}\, f(\{a\}) \subseteq B_i \,\} = \cap_{i\in I} \{\, a \mid f(\{a\}) \subseteq B_i \,\} = \cap_{i\in I} \overleftarrow{f}(B_i)$, and so, $\overleftarrow{f}$ is a complete $\cap$−morphism.

(ii) We prove: $\overleftarrow{f}$ is a sup-$\cup$−morphism.
The sup-$\cup$−morphism property is a consequence of the monotony of $\overleftarrow{f}$: $\forall i \in I : B_i \subseteq \cup_{j\in I} B_j$, so, $\overleftarrow{f}(B_i) \subseteq \overleftarrow{f}(\cup_{j\in I} B_j)$ and $\cup_i \overleftarrow{f}(B_i) \subseteq \overleftarrow{f}(\cup_{j\in I} B_j)$. To prove that $\overleftarrow{f}$ is not necessarily a $\cup$−morphism, consider the strict complete $\cup$−morphism f such that $f(\{a\}) = \{x, y\}$. Then, $\overleftarrow{f}(\{x\}) = \overleftarrow{f}(\{y\}) = \emptyset$ but $\overleftarrow{f}(\{x, y\}) = \{a\} \supsetneq \emptyset = \overleftarrow{f}(\{x\}) \cup \overleftarrow{f}(\{y\})$. To prove that $\overleftarrow{f}$ is not necessarily strict, consider the strict complete $\cup$−morphism f such that $f(\{a\}) = \emptyset$. Then, $\overleftarrow{f}(\emptyset) = \{a\} \supsetneq \emptyset$.

(iii) We prove: If f is a strict complete $\cup$−morphism, then $A \subseteq \overleftarrow{f}(B) \iff f(A) \subseteq B$.

$$
\begin{aligned}
& A \subseteq \overleftarrow{f}(B) \\
\iff\ & \forall a \in A : f(\{a\}) \subseteq B && \{\text{ def. of } \overleftarrow{\cdot} \,\} \\
\iff\ & \cup_{a\in A} f(\{a\}) \subseteq B \\
\iff\ & f(A) \subseteq B && \{\text{ strict complete } \cup\text{−morphism} \,\}
\end{aligned}
$$

(iv) We prove: $\overleftarrow{f \cup g} = \overleftarrow{f} \cap \overleftarrow{g}$.

$$\begin{array}{lll} & a \in (\overleftarrow{f \cup g})(B) & \\ \iff & f(\{a\}) \cup g(\{a\}) \subseteq B & \{\text{ def. of } \overleftarrow{\cdot}\ \} \\ \iff & f(\{a\}) \subseteq B \wedge g(\{a\}) \subseteq B & \\ \iff & a \in \overleftarrow{f}(B) \wedge a \in \overleftarrow{g}(B) & \{\text{ def. of } \overleftarrow{\cdot}\ \} \\ \iff & a \in (\overleftarrow{f} \cap \overleftarrow{g})(B) & \end{array}$$

(v) We prove: $\overleftarrow{f \cap g} \supseteq \overleftarrow{f} \cup \overleftarrow{g}$.

$$\begin{array}{lll} & a \in (\overleftarrow{f \cap g})(B) & \\ \iff & f(\{a\}) \cap g(\{a\}) \subseteq B & \{\text{ def. of } \overleftarrow{\cdot}\ \} \\ \impliedby & f(\{a\}) \subseteq B \vee g(\{a\}) \subseteq B & \\ \iff & a \in \overleftarrow{f}(B) \vee a \in \overleftarrow{g}(B) & \{\text{ def. of } \overleftarrow{\cdot}\ \} \\ \iff & a \in (\overleftarrow{f} \cup \overleftarrow{g})(B) & \end{array}$$

To prove that the equality does not necessarily hold, consider f and g such that $f(\{a\}) = \{x\}$ and $g(\{a\}) = \{y\}$. Then $(f \cap g)(\{a\}) = \emptyset$, and so, $a \in \overleftarrow{f \cap g}(\emptyset)$. However, $a \notin \overleftarrow{f}(\emptyset)$ and $a \notin \overleftarrow{g}(\emptyset)$.

(vi) We prove: If f is monotonic, then $\overleftarrow{f \circ g} \subseteq \overleftarrow{g} \circ \overleftarrow{f}$.

$$\begin{array}{lll} & a \in \overleftarrow{f \circ g}(B) & \\ \iff & (f \circ g)(\{a\}) \subseteq B & \{\text{ def. of } \overleftarrow{\cdot}\ \} \\ \implies & \forall b \in g(\{a\}) : f(\{b\}) \subseteq B & \{\text{ monotony of } f\ \} \\ \iff & \forall b \in g(\{a\}) : b \in \overleftarrow{f}(B) & \{\text{ def. of } \overleftarrow{\cdot}\ \} \\ \iff & g(\{a\}) \subseteq \overleftarrow{f}(B) & \\ \iff & a \in (\overleftarrow{g} \circ \overleftarrow{f})(B) & \{\text{ def. of } \overleftarrow{\cdot}\ \} \end{array}$$

(vii) We prove: If f is a strict complete $\cup$−morphism, then $\overleftarrow{f \circ g} = \overleftarrow{g} \circ \overleftarrow{f}$. When f is a strict complete $\cup$−morphism, it is monotonic. The proof of vi holds and we have moreover $(\forall b \in g(\{a\}) : f(\{b\}) \subseteq B) \implies (f \circ g)(\{a\}) = \cup \{\, f(\{b\}) \mid b \in g(\{a\}) \,\} \subseteq B$, which proves the equality.

(viii) We prove: $f \subseteq g \implies \overleftarrow{g} \subseteq \overleftarrow{f}$.

$$\begin{array}{lll} & a \in \overleftarrow{g}(B) & \\ \iff & g(\{a\}) \subseteq B & \{\text{ def. of } \overleftarrow{\cdot}\} \\ \implies & f(\{a\}) \subseteq B & \{ f \subseteq g \} \\ \iff & a \in \overleftarrow{f}(B) & \{\text{ def. of } \overleftarrow{\cdot}\} \end{array}$$

(ix) We prove: If f and g are strict complete $\cup$−morphisms, then $f \subseteq g \iff \overleftarrow{g} \subseteq \overleftarrow{f}$.
The $\implies$ part has been proved in viii. Suppose instead now that $\overleftarrow{g} \subseteq \overleftarrow{f}$, then we have on singletons:

$$\begin{array}{lll} & b \in f(\{a\}) & \\ \iff & \forall B : (f(\{a\}) \subseteq B \implies b \in B) & \\ \iff & \forall B : (a \in \overleftarrow{f}(B) \implies b \in B) & \{\text{ def. of } \overleftarrow{\cdot}\} \\ \implies & \forall B : (a \in \overleftarrow{g}(B) \implies b \in B) & \{ \overleftarrow{g} \subseteq \overleftarrow{f} \} \\ \iff & \forall B : (g(\{a\}) \subseteq B \implies b \in B) & \{\text{ def. of } \overleftarrow{\cdot}\} \\ \iff & b \in g(\{a\}) & \end{array}$$

So, $\forall a : f(\{a\}) \subseteq g(\{a\})$. When f and g are strict complete $\cup$−morphisms, the property lifts to arbitrary sets, and we have $f \subseteq g$. Note that this property implies that $\overleftarrow{g} = \overleftarrow{f} \implies f = g$.

(x) We prove: If g is a strict complete $\cup$−morphism, then $\overleftarrow{\lambda x.\mathrm{lfp}_x\,(\lambda z.z \cup f(z))} = \lambda y.\mathrm{gfp}_y(\lambda z.z \cap \overleftarrow{f}(z))$.
Let us note $h(z) \stackrel{\text{def}}{=} z \cup f(z)$. Then h is a strict extensive complete $\cup$−morphism, and any x is a pre-fixpoint of h. By Cousot's constructive version of Tarski's fixpoint theorem, we have $\forall x : \mathrm{lfp}_x\, h = \cup_{i\in\mathbb{N}}\, h^i(x)$. Moreover, each h^i is also a strict extensive complete $\cup$−morphism. We have:

$$\begin{array}{lll} & \mathrm{lfp}_x\, h \subseteq y & \\ \iff & \cup_{i\in\mathbb{N}}\, h^i(x) \subseteq y & \{\text{ Tarski's theorem }\} \\ \iff & \forall i \in \mathbb{N} : h^i(x) \subseteq y & \\ \iff & \forall i \in \mathbb{N} : x \subseteq \overleftarrow{h^i}(y) & \{\text{ property iii }\} \\ \iff & \forall i \in \mathbb{N} : x \subseteq \overleftarrow{h}^i(y) & \{\text{ property vii }\} \\ \iff & x \subseteq \cap_{i\in\mathbb{N}}\, \overleftarrow{h}^i(y) & \\ \iff & x \subseteq \mathrm{gfp}_y\, \overleftarrow{h} & \{\text{ Tarski's theorem }\} \end{array}$$

i.e., $\overleftarrow{\lambda x.\mathrm{lfp}_x\, h} = \lambda y.\mathrm{gfp}_y\, \overleftarrow{h}$. We conclude using property iv to get $\overleftarrow{h} = \overleftarrow{(\lambda x.x) \cup f} = \overleftarrow{\lambda x.x} \cap \overleftarrow{f} = (\lambda x.x) \cap \overleftarrow{f}$.

We note in passing that $\lambda x.\mathrm{lfp}_x\, h$ is a strict complete $\cup$−morphism, as the infinite join of strict complete $\cup$−morphisms. This property is useful when using $\lambda x.\mathrm{lfp}_x\, h$ as argument in the above properties that require it. □

We state without proof a few additional properties of interest (these are extremely simple and similar to the above ones):

- If f is extensive, then $\overleftarrow{f}$ is reductive.
- If f is reductive, then $\overleftarrow{f}$ is extensive.
- $\overleftarrow{\lambda A.A} = \lambda B.B$ (used in the proof of x).
- If f and g are monotonic and $f \circ g = g \circ f = \lambda x.x$, then $\overleftarrow{f} = g$ and $\overleftarrow{g} = f$.

The claim $\overleftarrow{\mathrm{post}} = \widetilde{\mathrm{pre}}$ stated in Sec. 3 is almost immediate:

$$\begin{aligned}
&\overleftarrow{\mathrm{post}}(B) \\
&= \{\, \sigma \in \Sigma \mid \mathrm{post}(\{\sigma\}) \subseteq B \,\} && \{\text{ definition of } \overleftarrow{\cdot} \,\} \\
&= \{\, \sigma \in \Sigma \mid \forall \sigma' : (\sigma, \sigma') \in \tau \implies \sigma' \in B \,\} && \{\text{ definition of post }\} \\
&= \widetilde{\mathrm{pre}}(B) && \{\text{ definition of } \widetilde{\mathrm{pre}} \,\}
\end{aligned}$$

Finally, we discuss briefly the use of these properties when deriving program semantics, hinted at the end of Sec. 3.

When the forward concrete invariance semantics is defined by structural induction on the syntax, we can apply the properties to derive its backward version, given that all the forward functions are complete $\cup$−morphism. Here are a few sample constructions:

- Sequences:
  we have $\tau\,\{\!|\, b_1; b_2 \,|\!\}\ \stackrel{\text{def}}{=}\ \tau\,\{\!|\, b_2 \,|\!\} \circ \tau\{\!|\, b_1 \,|\!\}$
  and so $\overleftarrow{\tau}\,\{\!|\, b_1; b_2 \,|\!\} = \overleftarrow{\tau}\,\{\!|\, b_1 \,|\!\} \circ \overleftarrow{\tau}\,\{\!|\, b_2 \,|\!\}$.
- Conditionals:
  we have $\tau\,\{\!|\, \texttt{if}\ (e)\ \{\, b_1 \,\}\ \texttt{else}\ \{\, b_2 \,\} \,|\!\}\, X \stackrel{\text{def}}{=} (\tau\,\{\!|\, b_1 \,|\!\} \circ \tau\{\!|\, e? \,|\!\})X \cup (\tau\,\{\!|\, b_2 \,|\!\} \circ \tau\{\!|\, \neg e? \,|\!\})X$
  and so $\overleftarrow{\tau}\,\{\!|\, \texttt{if}\ (e)\ \{\, b_1 \,\}\ \texttt{else}\ \{\, b_2 \,\} \,|\!\}\, X = (\overleftarrow{\tau}\,\{\!|\, e? \,|\!\} \circ \overleftarrow{\tau}\,\{\!|\, b_1 \,|\!\})X \cap (\overleftarrow{\tau}\,\{\!|\, \neg e? \,|\!\} \circ \overleftarrow{\tau}\,\{\!|\, b_2 \,|\!\})X$.
- Loops:
  we have $\tau\,\{\!|\, \texttt{while}\ (e)\ \{\, b \,\} \,|\!\}\, X \stackrel{\text{def}}{=} \tau\,\{\!|\, \neg e? \,|\!\}\, (\mathrm{lfp}_X\, \lambda Y.Y \cup (\tau\,\{\!|\, b \,|\!\} \circ \tau\{\!|\, e? \,|\!\})Y)$
  and so $\overleftarrow{\tau}\,\{\!|\, \texttt{while}\ (e)\ \{\, b \,\} \,|\!\}\, X = \mathrm{gfp}^{\leftarrow}_{\tau\{\neg e?\}X}\, \lambda Y.Y \cap (\overleftarrow{\tau}\,\{\!|\, e? \,|\!\} \circ \overleftarrow{\tau}\,\{\!|\, b \,|\!\})Y)$.

The forward semantics computes the set of all environments than can be seen at the end of the program as a function of the possible starting environments. The backward semantics, given a set of target environments at the end of the program, gives the set of initial environments such that all the executions that

reach the end of the program necessarily do so in a target environment.

B.4 Proof of Thm. 4.1

Proof. We prove: $\gamma_c(C \setminus \{\boldsymbol{a} \cdot \boldsymbol{x} \geq b\}) \subseteq \overleftarrow{\tau} \{\!\!\{ \boldsymbol{a} \cdot \boldsymbol{x} \geq b? \}\!\!\} \gamma_c(C)$.
Take $\rho \in \gamma_c(C \setminus \{\boldsymbol{a} \cdot \boldsymbol{x} \geq b\})$. Then, either $\boldsymbol{a} \cdot \rho(\boldsymbol{x}) \geq b$, in which case $\rho \in \gamma_c(C \cup \{\boldsymbol{a} \cdot \boldsymbol{x} \geq b\}) \subseteq \gamma_c(C)$, or $\boldsymbol{a} \cdot \rho(\boldsymbol{x}) < b$. In both cases, $\rho \in \gamma_c(C) \cup \{\, \rho \mid \boldsymbol{a} \cdot \rho(\boldsymbol{x}) < b \,\} = \overleftarrow{\tau} \{\!\!\{ \boldsymbol{a} \cdot \boldsymbol{x} \geq b? \}\!\!\} \gamma_c(C)$. □

We now justify the correctness of the two heuristics we proposed in Sec. 4.1, i.e., pre-processing the argument by P: (1) adding the constraint $\boldsymbol{a} \cdot \boldsymbol{x} \geq b$, and (2) adding rays r that satisfy $\tau \{\!\!\{ \boldsymbol{a} \cdot \boldsymbol{x} \geq b? \}\!\!\} \gamma_g(V_P, R_p \cup \{r\}) = \tau \{\!\!\{ \boldsymbol{a} \cdot \boldsymbol{x} \geq b? \}\!\!\} \gamma_g(V_P, R_P)$. Adding a constraint (1) is always sound as we under-approximate the argument which, by monotony of the concrete backward operator, under-approximates the result. We adding rays (2), we took care to only add environments ρ such that $\boldsymbol{a} \cdot \rho(\boldsymbol{x}) < b$, which are added by the concrete function $\overleftarrow{\tau} \{\!\!\{ \boldsymbol{a} \cdot \boldsymbol{x} \geq b? \}\!\!\}$ anyway, so the transformation is sound.

B.5 Proof of Thm. 4.2

Proof. We prove: If R is closed and convex, then $\overleftarrow{\tau} \{\!\!\{ V := ? \}\!\!\} R$ is either R or $\emptyset$.
Let us note $R' \stackrel{\text{def}}{=} \overleftarrow{\tau} \{\!\!\{ V := ? \}\!\!\} R = \{\, \rho \in \mathcal{E} \mid \forall v \in \mathbb{Q} : \rho[V \mapsto v] \in R \,\}$. We note that $R' \subseteq R$ (by choosing $v = \rho(V)$, we get $\rho \in R$).

We now prove that, if $R' \neq \emptyset$, then $R' = R$. Assume that $R' \neq \emptyset$ and, *ad absurdum*, that $R' \subsetneq R$, i.e., there exist $\rho, \rho' \in R$ such that $\rho \in R'$ but $\rho' \notin R'$. Thus, $\exists v' \in \mathbb{Q} : \rho'[V \mapsto v'] \notin R$. For any $\epsilon \in (0, 1]$, we now construct a point ρ'_ϵ in R that is at distance less than ϵ from $\rho'[V \mapsto v']$. We take ρ'_ϵ on the segment between $\rho' \in R$ and $\rho[V \mapsto M_\epsilon] \in R$: $\rho'_\epsilon \stackrel{\text{def}}{=} (1 - \alpha_\epsilon)\rho' + \alpha_\epsilon \rho[V \mapsto M_\epsilon]$, for some well-chosen M_ϵ and α_ϵ. More precisely, we choose $\alpha_\epsilon = \epsilon / \max\{\, 1, |\rho(W) - \rho'(W)| \mid W \neq V \,\}$ and $M_\epsilon = \rho'(V) + (v' - \rho'(V))/\alpha_\epsilon$. This implies $\forall W \neq V : |\rho'_\epsilon(W) - \rho'[V \mapsto v'](W)| = |((1-\alpha_\epsilon)\rho'(W) + \alpha_\epsilon \rho(W)) - \rho'(W)| = \alpha_\epsilon |\rho(W) - \rho'(W)| \leq \epsilon$. Moreover $|\rho'_\epsilon(V) - \rho'[V \mapsto v'](V)| = |((1 - \alpha_\epsilon)\rho'(V) + \alpha_\epsilon M_\epsilon) - v'| = |\rho'(V) - \alpha_\epsilon \rho'(V) + \alpha_\epsilon \rho'(V) + v' - \rho'(V)) - v'| = 0$. So, we indeed have $|\rho'_\epsilon - \rho'[V \mapsto v']|_\infty \leq \epsilon$. Finally, by convexity of R, we have $\rho'_\epsilon \in R$. We can thus construct a sequence of points in R that converges to $\rho'[V \mapsto v']$. As R is closed, this implies $\rho'[V \mapsto v'] \in R$, and so, our hypothesis $\rho' \notin R'$ is false. □

B.6 Proof of Thm. 4.3

Proof.

(i) We prove: If R is convex, then $\overleftarrow{\tau}\,\{\!|\, V := [a;b] \,|\!\}\, R = (\tau\,\{\!|\, V := ? \,|\!\} \circ (\tau\,\{\!|\, V := V - a \,|\!\} \cap \tau\{\!|\, V := V - b \,|\!\}) \circ \tau\{\!|\, V \geq a \wedge V \leq b? \,|\!\})R$.
By definition $\overleftarrow{\tau}\,\{\!|\, V := [a;b] \,|\!\}\, R = \{\, \rho \in \mathcal{E} \mid \forall y \in [a;b] : \rho[V \mapsto y] \in R \,\}$.

Assume that $\rho \in (\tau\,\{\!|\, V := ? \,|\!\} \circ (\tau\,\{\!|\, V := V - a \,|\!\} \cap \tau\{\!|\, V := V - b \,|\!\}) \circ \tau\{\!|\, V \geq a \wedge V \leq b? \,|\!\})R$. By definition of $\tau\{\!|\, V := ? \,|\!\}$, there exists some x such that $\rho[V \mapsto x] \in ((\tau\,\{\!|\, V := V - a \,|\!\} \cap \tau\{\!|\, V := V - b \,|\!\}) \circ \tau\{\!|\, V \geq a \wedge V \leq b? \,|\!\})R$. We have $\rho[V \mapsto x] \in (\tau\,\{\!|\, V := V - a \,|\!\} \circ \tau\{\!|\, V \geq a \wedge V \leq b? \,|\!\})R$, hence $\rho[V \mapsto x + a] \in \tau\,\{\!|\, V \geq a \wedge V \leq b? \,|\!\}\, R$. This implies in particular $x \geq 0$. Likewise, we have $\rho[V \mapsto x + b] \in \tau\,\{\!|\, V \geq a \wedge V \leq b? \,|\!\}\, R$, and so $x \leq 0$. We deduce that $x = 0$ and $\rho[V \mapsto a], \rho[V \mapsto b] \in \tau\,\{\!|\, V \geq a \wedge V \leq b? \,|\!\}\, R \subseteq R$. By convexity of R, $\forall y \in [a;b] : \rho[V \mapsto y] \in R$. Hence, $\rho \in \overleftarrow{\tau}\,\{\!|\, V := [a;b] \,|\!\}\, R$.

Conversely, suppose that $\rho \in \overleftarrow{\tau}\,\{\!|\, V := [a;b] \,|\!\}\, R$. Then, by definition, $\rho[V \mapsto a], \rho[V \mapsto b] \in R$. Obviously, $\rho[V \mapsto a], \rho[V \mapsto b] \in \tau\{\!|\, V \geq a \wedge V \leq b? \,|\!\})R$. As a consequence $\rho[V \mapsto 0] \in ((\tau\,\{\!|\, V := V - a \,|\!\} \cap \tau\{\!|\, V := V - b \,|\!\}) \circ \tau\{\!|\, V \geq a \wedge V \leq b? \,|\!\})R$. We deduce that $\rho \in (\tau\,\{\!|\, V := ? \,|\!\} \circ (\tau\,\{\!|\, V := V - a \,|\!\} \cap \tau\{\!|\, V := V - b \,|\!\}) \circ \tau\{\!|\, V \geq a \wedge V \leq b? \,|\!\})R$.

Note that we can safely under-approximate $\overleftarrow{\tau}\{\!|\, V := X \,|\!\}$ for any constant set X with $\overleftarrow{\tau}\{\!|\, V := [\min X; \max X] \,|\!\}$.

(ii) We prove: If R is convex, then $\overleftarrow{\tau}\,\{\!|\, V := V + [a;b] \,|\!\}\, R = \tau\,\{\!|\, V := V - a \,|\!\}\, R \cap \tau\,\{\!|\, V := V - b \,|\!\}\, R$.

$$
\begin{aligned}
&\overleftarrow{\tau}\,\{\!|\, V := V + [a;b] \,|\!\}\, R \\
&= \{\, \rho \mid \forall x \in [a;b] : \rho[V \mapsto \rho(V) + x] \in R \,\} \\
&= \{\, \rho \mid \forall x \in \{a,b\} : \rho[V \mapsto \rho(V) + x] \in R \,\} \\
&= \{\, \rho \mid \rho[V \mapsto \rho(V) + a] \in R \,\} \cap \{\, \rho \mid \rho[V \mapsto \rho(V) + b] \in R \,\} \\
&= \tau\,\{\!|\, V := V - a \,|\!\}\, R \cap \tau\,\{\!|\, V := V - b \,|\!\}\, R
\end{aligned}
$$

The fact that $\forall x \in [a;b] : \rho[V \mapsto \rho(V) + x] \in R \iff \forall x \in \{a,b\} : \rho[V \mapsto \rho(V) + x] \in R$ comes form the hypothesis that R is convex.

(iii) We prove: $\overleftarrow{\tau}\,\{\!|\, V := W \,|\!\} = \tau\,\{\!|\, V := ? \,|\!\} \circ \tau\{\!|\, V = W? \,|\!\}$.
By definition, $\overleftarrow{\tau}\,\{\!|\, V := W \,|\!\}\, R = \{\, \rho \in \mathcal{E} \mid \rho[V \mapsto \rho(W)] \in R \,\}$.

Take $\rho \in (\tau\,\{\!|\, V := ? \,|\!\} \circ \tau\{\!|\, V = W? \,|\!\})R$. Then, by definition, $\exists x : \rho[V \mapsto x] \in \tau\,\{\!|\, V = W? \,|\!\}\, R$, i.e., $\rho[V \mapsto x] \in R$ and $\rho(W) = \rho[V \mapsto x](V) = x$. We deduce that $\rho[V \mapsto \rho(W)] = \rho[V \mapsto x] \in R$, and so, $\rho \in \overleftarrow{\tau}\,\{\!|\, V := W \,|\!\}\, R$.

Take now $\rho \in \overleftarrow{\tau}\,\{\!|\, V := W \,|\!\}\, R$. Then, $\rho[V \mapsto \rho(W)] \in R$. As obviously $\rho[V \mapsto \rho(W)]$ satisfies the constraint $V = W$, we have $\rho[V \mapsto \rho(W)] \in \tau\,\{\!|\, V = W? \,|\!\}\, R$. Then $\rho \in (\tau\,\{\!|\, V := ? \,|\!\} \circ \tau\{\!|\, V = W? \,|\!\})R$.

(iv) If $V := e$ is invertible then $\overleftarrow{\tau}\,\{\!|\, V := e \,|\!\} = \tau\{\!|\, V := e^{-1} \,|\!\}$.

By definition of invertible, $\tau \{\!|\, V := e \,|\!\} \circ \tau \{\!|\, V := e^{-1} \,|\!\} = \tau \{\!|\, V := e^{-1} \,|\!\} \circ \tau \{\!|\, V := e \,|\!\} = \lambda x.x$. We use the fact that $f \circ g = g \circ f = \lambda x.x$ for monotonic f and g implies that $\overleftarrow{f} = g$, as stated before and proved as follows:

$$
\begin{array}{lll}
 & a \in \overleftarrow{f}(B) & \\
\iff & f(\{a\}) \subseteq B & \{\text{ def. } \overleftarrow{\cdot} \} \\
\implies & (g \circ f)(\{a\}) \subseteq g(B) & \{\text{ monotony of } g \} \\
\iff & a \in g(B) & \{\, g \circ f = \lambda x.x \,\}
\end{array}
$$

$$
\begin{array}{lll}
 & a \in \overleftarrow{f}(B) & \\
\iff & f(\{a\}) \subseteq B & \{\text{ def. } \overleftarrow{\cdot} \} \\
\iff & (f \circ g \circ f)(\{a\}) \subseteq (f \circ g)(B) & \{\, f \circ g = \lambda x.x \,\} \\
\impliedby & (g \circ f)(\{a\}) \subseteq g(B) & \{\text{ monotony of } f \} \\
\iff & a \in g(B) & \{\, g \circ f = \lambda x.x \,\}
\end{array}
$$

□

B.7 Proof of Thm. 4.4

Proof. We prove: $\underline{\triangledown}$ is a lower widening.
We first prove that $A \mathbin{\underline{\triangledown}} B \subseteq A \cap B$, i.e., using the generator representation $\gamma_g(V_{A \underline{\triangledown} B}, R_{A \underline{\triangledown} B}) \subseteq \gamma_g(V_A, R_A) \cap \gamma_g(V_B, R_B)$. We first note that $V_{A \underline{\triangledown} B} \subseteq V_A$ and $R_{A \underline{\triangledown} B} \subseteq R_A$, which implies directly $\gamma_g(V_{A \underline{\triangledown} B}, R_{A \underline{\triangledown} B}) \subseteq \gamma_g(V_A, R_A)$. Consider now a point $x \in \gamma_g(V_{A \underline{\triangledown} B}, R_{A \underline{\triangledown} B})$. By definition of γ_g, x can be written as $x = \sum_{v \in V_{A \underline{\triangledown} B}} \alpha_v v + \sum_{r \in R_{A \underline{\triangledown} B}} \beta_r r$ for some $\alpha_v, \beta_r \geq 0$ such that $\sum_v \alpha_v = 1$. As $V_{A \underline{\triangledown} B} \subseteq B$, by convexity of B, $\sum_{v \in V_{A \underline{\triangledown} B}} \alpha_v v \in B$. Moreover, for each $r \in R_{A \underline{\triangledown} B}$, we have $B \oplus \beta_r r \subseteq B \oplus \mathbb{R}^+ r = B$, so, $x \in B$.

The termination follows simply from the fact that, in any sequence $X_{i+1} = X_i \mathbin{\underline{\triangledown}} Y_{i+1}$, the set of generators in X_i decreases. Whenever $V_{X_i} = \emptyset$, the corresponding polyhedron is empty. □

B.8 Proof of Thm. 4.5

Proof.

(i) We prove: If $e \sqsubseteq_D f$, then $\overleftarrow{\tau} \{\!|\, V := e \,|\!\} R \supseteq (\overleftarrow{\tau} \{\!|\, V := f \,|\!\} R) \cap D$.

$$
\begin{aligned}
& \overleftarrow{\tau}\, \{\!|\, V := e \,|\!\}\, R \\
\supseteq\ & (\overleftarrow{\tau}\, \{\!|\, V := e \,|\!\}\, R) \cap D \\
=\ & \{\, \rho \in D \mid \forall v \in [\![\, e \,]\!]\rho : \rho[V \mapsto v] \in R \,\} && \{\, \text{def. of } \overleftarrow{\cdot} \,\} \\
\supseteq\ & \{\, \rho \in D \mid \forall v \in [\![\, f \,]\!]\rho : \rho[V \mapsto v] \in R \,\} && \{\, e \sqsubseteq_D f \,\} \\
=\ & (\overleftarrow{\tau}\, \{\!|\, V := f \,|\!\}\, R) \cap D
\end{aligned}
$$

(ii) We prove: If $e \sqsubseteq_D f$, then $\overleftarrow{\tau}\, \{\!|\, e? \,|\!\}\, R \supseteq (\overleftarrow{\tau}\, \{\!|\, f? \,|\!\}\, R) \cap D$.
As we now manipulate boolean expressions instead of numeric ones, we consider the evaluation function $[\![\, e \,]\!] : \mathcal{E} \to \mathcal{P}(\{\mathrm{t}, \mathrm{f}\})$ that denotes whether e may hold (t) or may not hold (f) in an environment in $\mathcal{E}$. Then:

$$
\begin{aligned}
& \overleftarrow{\tau}\, \{\!|\, e? \,|\!\}\, R \\
=\ & R \cup \{\, \rho \in \mathcal{E} \mid t \notin [\![\, e \,]\!]\rho \,\} && \{\, \text{def. of } \overleftarrow{\tau}\, \{\!|\, e? \,|\!\} \,\} \\
\supseteq\ & R \cup \{\, \rho \in D \mid t \notin [\![\, e \,]\!]\rho \,\} \\
\supseteq\ & R \cup \{\, \rho \in D \mid t \notin [\![\, f \,]\!]\rho \,\} && \{\, e \sqsubseteq_D f \,\} \\
\supseteq\ & D \cap (R \cup \{\, \rho \mid t \notin [\![\, f \,]\!]\rho \,\}) \\
=\ & D \cap (\overleftarrow{\tau}\, \{\!|\, f? \,|\!\}\, R) && \{\, \text{def. of } \overleftarrow{\tau}\, \{\!|\, f? \,|\!\} \,\}
\end{aligned}
$$

□