Backward Analysis via Over-Approximate Abstraction and Under-Approximate Subtraction

Alexey Bakhirkin[1], Josh Berdine[2], and Nir Piterman[1]

[1] University of Leicester, Department of Computer Science
[2] Microsoft Research

Abstract. We propose a novel approach for computing weakest liberal safe preconditions of programs. The standard approaches, which call for either under-approximation of a greatest fixed point, or complementation of a least fixed point, are often difficult to apply successfully. Our approach relies on a different decomposition of the weakest precondition of loops. We exchange the greatest fixed point for the computation of a least fixed point above a recurrent set, instead of the bottom element. Convergence is achieved using over-approximation, while in order to maintain soundness we use an under-approximating logical subtraction operation. Unlike general complementation, subtraction more easily allows for increased precision in case its arguments are related. The approach is not restricted to a specific abstract domain and we use it to analyze programs using the abstract domains of intervals and of 3-valued structures.

1 Introduction

Forward static analyses usually compute program invariants which hold of executions starting from given initial conditions, e.g., over-approximations of reachable states. Conversely, backward static analyses for universal properties compute program invariants which ensure given assertions hold of all executions, e.g., under-approximations of safe states. Forward analysis of programs has been a notable success, while such backward analysis has seen much less research and is done less frequently (a notable example is [17]).

The standard formulation of forward analyses involves over-approximating a least fixed point of a recursive system of equations (transformers) that over-approximate the forward semantics of commands. Conversely, backward analyses for universal properties usually involve under-approximating a greatest fixed point of under-approximate equations.

The over-approximating abstractions used by forward analyses are far more common and well-developed than the under-approximations used by backward analyses. One approach to under-approximation is via over-approximate abstraction and under-approximate complementation $\overline{(\cdot)}$. For example, lower widening $p \mathbin{\underline{\triangledown}} q$ may be seen as $\overline{\overline{p} \mathbin{\triangledown} \overline{q}}$. However, computing the complement is, in many cases, infeasible or impractical (e.g., for 3-valued structures [22], separation logic [8], or polyhedra [12]).

Here, we suggest an alternative backward analysis approach that uses least fixed-point approximation, and an *under-approximate logical subtraction* operation in lieu

of complementation. (Logical subtraction can also be understood as *and with complement* or *not implies*.) We show how to extend a computation of a recurrent set of a program with a least fixed-point approximation to obtain an under-approximation of the safe states from which *no* execution can lead to a failure (such as violating an assertion, dividing by zero, or dereferencing a dangling-pointer – i.e., an event that causes program execution to immediately abort and signal an error). Soundness is ensured by subtracting an over-approximation of the unsafe states.

Using subtraction instead of complementation has several advantages. First, it is easier to define in power set domains for which complementation can be hard or impractical. Second, as the approximations of safe and unsafe states are the results of analyzing the same code, they are strongly related and so subtraction may be more precise than a general under-approximate complementation.

Our approach is not restricted to a specific abstract domain and we use it to analyze numeric examples (using the domain of intervals) and examples coming from shape analysis (using the domain of 3-valued structures).

2 Preliminaries

Let $\mathcal{U}$ denote the set of program *memory* states and $\epsilon \notin \mathcal{U}$ a *failure* state. The concrete domain for our analysis is the power set $\mathcal{P}(\mathcal{U})$ ordered by $\subseteq$, with least element $\varnothing$, greatest element $\mathcal{U}$, join $\cup$, and meet $\cap$.

We introduce an abstract domain $\mathcal{D}$ (with $\sqsubseteq$, $\bot$, $\top$, $\sqcup$, and $\sqcap$) and a concretization function $\gamma : \mathcal{D} \to \mathcal{P}(\mathcal{U})$. For an element of an abstract domain, $d \in \mathcal{D}$, $\gamma(d)$ is the set of states represented by it. For example, for a program with two variables x and y, an element of the interval domain $d = \langle x : [1;2], y : [3;4] \rangle$ represents all states satisfying $(1 \leq x \leq 2) \wedge (3 \leq y \leq 4)$, i.e., $\gamma(d) = \{(x, y) \mid 1 \leq x \leq 2 \wedge 3 \leq y \leq 4\}$.

For a lattice $\mathcal{L}$, we define *complementation* as a function $\overline{(\cdot)} : \mathcal{L} \to \mathcal{L}$ such that for every $l \in \mathcal{L}$, $\gamma(\overline{l}) \cap \gamma(l) = \varnothing$ (i.e., they represent disjoint sets of states – but we do not require that $\gamma(\overline{l}) \cup \gamma(l) = \mathcal{U}$). For example, if $d \in \mathcal{D}$ over-approximates the unsafe states, then $\overline{d}$ under-approximates the safe states. For our concrete domain $\mathcal{P}(\mathcal{U})$ (and similarly, for every power set of atomic elements), we can use standard set-theoretic complement: $\overline{S} = \mathcal{U} \setminus S$.

We define *subtraction* as a function $(\cdot - \cdot) : \mathcal{L} \to \mathcal{L} \to \mathcal{L}$ such that for $l_1, l_2 \in \mathcal{L}$ we have $\gamma(l_1 - l_2) \subseteq \gamma(l_1)$ and $\gamma(l_1 - l_2) \cap \gamma(l_2) = \varnothing$. For example, given a domain $\mathcal{D}$, we can define subtraction for the power set domain $\mathcal{P}(\mathcal{D})$ as

$$D_1 - D_2 = \{d_1 \in D_1 \mid \forall\, d_2 \in D_2.\ \gamma(d_1) \cap \gamma(d_2) = \varnothing\} \tag{1}$$

This way, subtraction can be defined in e.g., the domain of 3-valued structures that does not readily support complementation. We claim that a useful subtraction is often easier to define than a useful complementation. We also note that for every $l_0 \in \mathcal{L}$, the function $\lambda l.(l_0 - l)$ is a complementation. However, for a given l, the accuracy of this complement depends on the actual choice of l_0.

2.1 Programming Language Syntax and Semantics

We consider a simple structured programming language. Given a set of *atomic statements* A ranged over by a, statements C of the language are constructed as follows:

$$\begin{array}{lll} C ::= a & & \text{atomic statement} \\ \quad \mid C_1 \,;\, C_2 & & \text{sequential composition: executes } C_1 \text{ and then } C_2 \\ \quad \mid C_1 + C_2 & & \text{branch: non-deterministically branches to either } C_1 \text{ or } C_2 \\ \quad \mid C^* & & \text{loop: iterated sequential composition of } \geq 0 \text{ copies of } C \end{array}$$

We assume A contains: the empty statement `skip`, an assertion statement $\texttt{assert}\,\varphi$ (for a state formula φ), and an assumption statement $[\varphi]$. Informally, an assertion immediately aborts the execution and signals an error if φ is not satisfied, and we consider that there are no *valid* executions violating assumptions. Standard conditionals $\texttt{if}(\varphi)\ C_1\ \texttt{else}\ C_2$ can be expressed by $([\varphi]; C_1) + ([\neg\varphi]; C_2)$. Similarly, loops $\texttt{while}(\varphi)\ C$ can be expressed by $([\varphi]; C)^* \,;\, [\neg\varphi]$.

A state formula φ denotes a set of non-failure states $[\![\varphi]\!] \subseteq \mathcal{U}$ that satisfy φ. The semantics of a statement C is a relation $[\![C]\!] \subseteq \mathcal{U} \times (\mathcal{U} \cup \{\epsilon\})$. For $s, s' \in \mathcal{U}$, $[\![C]\!](s, s')$ means that executing C in state s *may* change the state to s'. Then, $[\![C]\!](s, \epsilon)$ means that s is unsafe: executing C from state s may result in failure (may cause the program to immediately abort). Let $\Delta_{\mathcal{U}}$ be the diagonal relation on states $\Delta_{\mathcal{U}} = \{(s, s) \mid s \in \mathcal{U}\}$. Let composition of relations in $\mathcal{U} \times (\mathcal{U} \cup \{\epsilon\})$ be defined as $S \fatsemi R = (R \cup \{(\epsilon, \epsilon)\}) \circ S$ where $\circ$ is standard composition of relations. Fixed points in $\mathcal{U} \times (\mathcal{U} \cup \{\epsilon\})$ are with respect to the subset order, where $\mathrm{lfp}\, \lambda X.\, F(X)$ denotes the least fixed point of $\lambda X.\, F(X)$, and similarly, $\mathrm{gfp}\, \lambda X.\, F(X)$ denotes the greatest fixed point of $\lambda X.\, F(X)$. For an atomic statement a, we assume that $[\![a]\!]$ is a predefined left-total relation, and the semantics of other statements is defined as follows:

$$\begin{array}{rlrl} [\![\texttt{skip}]\!] &= \Delta_{\mathcal{U}} & [\![C_1 \,;\, C_2]\!] &= [\![C_1]\!] \fatsemi [\![C_2]\!] \\ [\![[\varphi]]\!] &= \{(s, s) \mid s \in [\![\varphi]\!]\} & [\![C_1 + C_2]\!] &= [\![C_1]\!] \cup [\![C_2]\!] \\ [\![\texttt{assert}\,\varphi]\!] &= \{(s, s) \mid s \in [\![\varphi]\!]\} \cup & [\![C^*]\!] &= \mathrm{lfp}\, \lambda X.\, \Delta_{\mathcal{U}} \cup ([\![C]\!] \fatsemi X) \\ & \quad \{(s, \epsilon) \mid s \in \mathcal{U} \wedge s \notin [\![\varphi]\!]\} & & \end{array}$$

Note that the assumption that atomic statements denote left-total relations excludes statements that affect control flow such as `break` or `continue`. In what follows, we constrain considered programs in the following way. Programs cannot have nested loops and assumption statements $[\varphi]$ are only allowed to appear at the start of branches and at the entry and exit of loops (they cannot be used as normal atomic statements):

$$C ::= a \mid C_1 \,;\, C_2 \mid ([\varphi] \,;\, C_1) + ([\psi] \,;\, C_2) \mid ([\psi] \,;\, C)^* \,;\, [\varphi]$$

We require that for branches and loops, $\varphi \vee \psi = 1$ (i.e., $[\![\varphi]\!] \cup [\![\psi]\!] = \mathcal{U}$). That is, for a loop-free statement, the domain of its semantics is $\mathcal{U}$. We also require that the language of state formulas is closed under negation.

2.2 Fixed-Point Characterizations of Safe and Unsafe States

Given a statement C and a set of states $S \subseteq \mathcal{U}$, we define:

- $pre(C, S) = \{s \in \mathcal{U} \mid \exists s' \in S.\ [\![C]\!](s, s')\}$. The states that may lead to S after executing C.
- $fail(C) = \{s \in \mathcal{U} \mid [\![C]\!](s, \epsilon)\}$. The *unsafe* states: those that may cause C to fail.
- $wp(C, S) = \{s \in \mathcal{U} \mid \forall s' \in \mathcal{U} \cup \{\epsilon\}.\ [\![C]\!](s, s') \Rightarrow s' \in S\}$. The *weakest liberal precondition* that ensures safety [7] – safe states that must lead to S if execution of the statement terminates.

We abbreviate $pre(C, S) \cup fail(C)$ to $pre{+}fail(C, S)$.

Lemma 1. *For a statement C and a set of states $S \subseteq \mathcal{U}$,*
$wp(C, S) = \mathcal{U} \setminus pre{+}fail(C, \mathcal{U} \setminus S)$.

The proof is a direct calculation based on the definitions. See the companion technical report [3] for proofs.

For a program C, our goal is to compute (an under-approximation of) $wp(C, \mathcal{U})$, and (an over-approximation of) its complement $fail(C)$. If we are interested in termination with specific postcondition φ, we add an $\texttt{assert}\,\varphi$ statement to the end of the program. We characterize these sets (as is standard [9,10]) as solutions to two functionals P and N that associate a statement C and a set of states S (resp., V) $\subseteq \mathcal{U}$ with a predicate $P(C, S)$, resp., $N(C, V)$. $P(C, S)$ (the *positive side*) denotes the states that must either lead to successful termination in S or cause non-termination, and $N(C, V)$ (the *negative side*) denotes the states that may lead to failure or termination in V.

$$
\begin{array}{ll}
P(a, S) = wp(a, S) & N(a, V) = pre{+}fail(a, V) \\
P([\varphi], S) = [\![\neg\varphi]\!] \cup S & N([\varphi], V) = [\![\varphi]\!] \cap V \\
P(\texttt{assert}\,\varphi, S) = [\![\varphi]\!] \cap S & N(\texttt{assert}\,\varphi, V) = [\![\neg\varphi]\!] \cup V \\
P(C_1; C_2, S) = P(C_1, P(C_2, S)) & N(C_1; C_2, V) = N(C_1, N(C_2, V)) \\
P(C_1 + C_2, S) = P(C_1, S) \cap P(C_2, S) & N(C_1 + C_2, V) = N(C_1, V) \cup N(C_2, V) \\
P(C^*, S) = \mathrm{gfp}\,\lambda X.\, S \cap P(C, X) & N(C^*, V) = \mathrm{lfp}\,\lambda Y.\, V \cup N(C, Y)
\end{array}
$$

Lemma 2. *For a statement C and set of states $S \subseteq \mathcal{U}$, $P(C, S) = \mathcal{U} \setminus N(C, \mathcal{U} \setminus S)$.*

The proof is by structural induction.

Lemma 3. *For a statement C and sets of states $S, V \subseteq \mathcal{U}$, $P(C, S) = wp(C, S)$, and $N(C, V) = pre{+}fail(C, V)$.*

The proof is by structural induction, relying on continuity of *pre+fail*.

3 Least Fixed-Point Characterization of Safe States

The direct solution of the positive side is by under-approximating a greatest fixed point. This can be problematic since most domains are geared towards over-approximating least fixed points. Hence, we are not going to approximate the greatest fixed point for the positive side directly. Instead, we restate the problem for loops such that the resulting characterization leads to a least fixed point computation where termination is ensured by using an appropriate over-approximate abstraction.

In this section, we focus on the looping statement:

$$C_{\text{loop}} = ([\psi]\ ;\ C_{\text{body}})^*\ ;\ [\varphi] \tag{2}$$

where C_{body} is the *loop body*; if ψ holds the execution may enter the loop body; and if φ holds the execution may exit the loop. To simplify the presentation, in what follows, we assume that the semantics of C_{body} is directly known. Since C_{body} is itself loop-free, $[\![C_{\text{body}}]\!]$ does not induce fixed points, and the transformers for the loop body can be obtained, e.g., by combining the transformers for its sub-statements.

3.1 Recurrent Sets

We reformulate the characterizations of safe states in terms of least fixed points with the use of recurrent sets. For the loop in (2), an *existential recurrent set* is a set $R_\exists$, s.t.

$$\begin{aligned} &R_\exists \subseteq [\![\psi]\!] \\ &\forall s \in R_\exists.\ \exists s' \in R_\exists.\ [\![C_{\text{body}}]\!](s, s') \end{aligned}$$

These are states that may cause non-termination (i.e., cause the computation to stay inside the loop forever). For the loop in (2), a *universal recurrent set* is a set $R_\forall$, s.t.

$$\begin{aligned} &R_\forall \subseteq [\![\neg\varphi]\!] \\ &\forall s \in R_\forall.\ \big(\forall s' \in \mathcal{U} \cup \{\epsilon\}.\ [\![C_{\text{body}}]\!](s, s') \Rightarrow s' \in R_\forall\big) \end{aligned}$$

These are states that must cause non-termination. For practical reasons discussed later in Sect. 4.2, we *do not* require these sets to be maximal.

Lemma 4. *For the loop* $C_{\text{loop}} = ([\psi]\ ;\ C_{\text{body}})^*\ ;\ [\varphi]$*, and a set of states* $S \subseteq \mathcal{U}$

$$R_\forall \subseteq P(C_{\text{loop}}, S) \qquad\qquad R_\exists \smallsetminus N(C_{\text{loop}}, \mathcal{U} \smallsetminus S) \subseteq P(C_{\text{loop}}, S)$$

For $R_\forall$, the proof correlates universal recurrence and *wp*, relying on monotonicity of $P(C_{\text{loop}}, \cdot)$. For $R_\exists$, the result follows from Lemma 2.

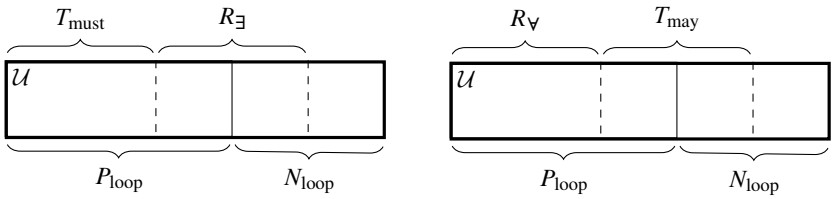
3.2 Positive Least Fixed Point via Recurrent Sets

We begin with an informal explanation of how we move from a greatest fixed point formulation to a least fixed point one. Observe that for the loop in (2), the positive and negative sides (following the definition in Sect. 2.2) are characterized by:

$$\begin{aligned} P(C_{\text{loop}}, S) &= \operatorname{gfp} \lambda X.\ ([\![\neg\varphi]\!] \cup S) \cap \big([\![\neg\psi]\!] \cup P(C_{\text{body}}, X)\big) \\ N(C_{\text{loop}}, V) &= \operatorname{lfp} \lambda Y.\ ([\![\varphi]\!] \cap V) \cup \big([\![\psi]\!] \cap N(C_{\text{body}}, Y)\big) \end{aligned} \tag{3}$$

Then, since loops only occur at the top level, a program C_{prg} that contains the loop C_{loop} can be expressed as $C_{\text{init}}\ ;\ C_{\text{loop}}\ ;\ C_{\text{rest}}$ (where C_{init} or C_{rest} may be `skip`). Let:

- $P_{\text{rest}} = P(C_{\text{rest}}, \mathcal{U})$ – the safe states of the loop's continuation.
- $N_{\text{rest}} = N(C_{\text{rest}}, \varnothing)$ – states that may cause failure of the loop's continuation. Note that $N_{\text{rest}} = \mathcal{U} \smallsetminus P_{\text{rest}}$.



(a) Partitioning with existential recurrence. (b) Partitioning with universal recurrence.

Fig. 1: Partitioning of the states at the loop entry.

- $P_{\text{loop}} = P(C_{\text{loop}}, P_{\text{rest}})$ – the safe states of the loop and its continuation.
- $N_{\text{loop}} = N(C_{\text{loop}}, N_{\text{rest}})$ – states that may cause failure of the loop or its continuation. Note that $N_{\text{loop}} = \mathcal{U} \setminus P_{\text{loop}}$.

For the loop in (2), Fig. 1 shows how the states entering the loop can be partitioned. In the figure, by T_{must}, we denote the states that must cause successful termination *of the loop* (in a state belonging to P_{rest}), and by T_{may}, we denote states that may cause successful termination.

Fig. 1a shows that the positive side for the loop in (2) can be partitioned into the following two parts:

- $R_\exists \setminus N_{\text{loop}}$ – states that may cause non-termination but may not fail;
- T_{must} – states that must cause successful termination of the loop.

T_{must} can be characterized as the least fixed point:

$$T_{\text{must}} = \text{lfp}\, \lambda X.\, (\llbracket \neg\psi \rrbracket \cap P_{\text{rest}}) \cup \Big(((\llbracket \psi \rrbracket \cap P_{\text{rest}}) \cup \llbracket \neg\varphi \rrbracket) \cap wp(C_{\text{body}}, X) \Big)$$

Intuitively, the states in $\llbracket \neg\psi \rrbracket \cap P_{\text{rest}}$ cause the loop to immediately terminate (such that the rest of the program does not fail), those in $((\llbracket \psi \rrbracket \cap P_{\text{rest}}) \cup \llbracket \neg\varphi \rrbracket) \cap wp(C_{\text{loop}}, \llbracket \neg\psi \rrbracket \cap P_{\text{rest}})$ can make one iteration through the loop, and so on.

Fig. 1b shows that the positive side can also be partitioned in another way:

- $R_\forall$ – states that must cause non-termination of the loop;
- $T_{\text{may}} \setminus N_{\text{loop}}$ – states that may cause successful termination but may not fail.

In a way similar to [10], T_{may} can be characterized as the least fixed point:

$$T_{\text{may}} = \text{lfp}\, \lambda X.\, (\llbracket \varphi \rrbracket \cap P_{\text{rest}}) \cup \big(\llbracket \psi \rrbracket \cap pre(C_{\text{body}}, X) \big)$$

Intuitively, from states $\llbracket \varphi \rrbracket \cap P_{\text{rest}}$, the loop *may* immediately terminate in a state safe for C_{rest}, from states $\llbracket \psi \rrbracket \cap pre(C_{\text{body}}, \llbracket \varphi \rrbracket \cap P_{\text{rest}})$ the loop may make one iteration and terminate, and so on. From this, it can be shown that

$$T_{\text{may}} \setminus N_{\text{loop}} = \text{lfp}\, \lambda X.\, \big((\llbracket \varphi \rrbracket \cap P_{\text{rest}}) \setminus N_{\text{loop}} \big) \cup \big((\llbracket \neg\varphi \rrbracket \cap pre(C_{\text{body}}, X)) \setminus N_{\text{loop}} \big)$$

We replace ψ with $\neg\varphi$, since the states in $\llbracket \psi \rrbracket \cap \llbracket \varphi \rrbracket \cap pre(C_{\text{body}}, X)$ are either already included in the first disjunct (if belonging to P_{rest}), or are unsafe and removed by subtraction.

Following these least fixed point characterizations, we re-express the equation for the positive side of the loop (3) using the existential recurrent set $R_\exists$ as follows, where $N = N(C_{\text{loop}}, \mathcal{U} \setminus S)$:

$$P^\exists(C_{\text{loop}}, S) = \text{lfp}\, \lambda X.\, (R_\exists \setminus N) \cup (\llbracket \neg\psi \rrbracket \cap S) \cup \Big(((\llbracket \psi \rrbracket \cap S) \cup \llbracket \neg\varphi \rrbracket) \cap wp(C_{\text{body}}, X) \Big) \tag{4}$$

or using the universal recurrent set $R_\forall$ as follows:

$$P^\forall(C_{\text{loop}}, S) = \text{lfp}\, \lambda X.\, R_\forall \cup \big((\llbracket \varphi \rrbracket \cap S) \setminus N \big) \cup \Big(\big(\llbracket \neg\varphi \rrbracket \cap pre(C_{\text{body}}, X) \big) \setminus N \Big) \tag{5}$$

Theorem 1. *The alternative characterizations of the positive side of the loop:* (4) *and* (5) *– under-approximate the original characterization* (3)*. That is, for a set* $S \subseteq \mathcal{U}$*,*

$$P^\exists(C_{\text{loop}}, S) \subseteq P(C_{\text{loop}}, S) \qquad\qquad P^\forall(C_{\text{loop}}, S) \subseteq P(C_{\text{loop}}, S)$$

4 Approximate Characterizations

In Sects. 2.2 and 3.2, we characterized both the negative and the positive sides as least fixed points. For the negative side, our goal is to over-approximate the least fixed point, and we can do that using standard tools. That is, we move to an abstract domain $\mathcal{D}(\sqsubseteq, \bot, \top, \sqcup, \sqcap, \triangledown)$ where widening $\triangledown$ and join $\sqcup$ may coincide for domains that do not allow infinite ascending chains. For the positive side our goal is to under-approximate the least fixed point, and to do so, we build an increasing chain of its approximations and use the previously computed negative side and subtraction to ensure soundness.

As before, since we do not allow nested loops, we assume that abstract transformers for loop bodies are given. For a loop-free statement C and $d \in \mathcal{D}$, we assume: over- and under-approximating transformers $pre^\sharp(C, d)$ and $wp^\flat(C, d)$, over-approximating operation $fail^\sharp(C)$; and for assumption statements $[\varphi]$: under- and over-approximate transformers $[\varphi, d]^\flat$ and $[\varphi, d]^\sharp$ such that:

$$\gamma(pre^\sharp(C, d)) \supseteq pre(C, \gamma(d)) \qquad \gamma(fail^\sharp(C)) \supseteq fail(C)$$
$$\gamma(wp^\flat(C, d)) \subseteq wp(C, \gamma(d)) \qquad \gamma([\varphi, d]^\flat) \subseteq \llbracket \varphi \rrbracket \cap \gamma(d) \subseteq \gamma([\varphi, d]^\sharp)$$

We abbreviate $[\varphi, \top]^\flat$ to $[\varphi]^\flat$ and $[\varphi, \top]^\sharp$ to $[\varphi]^\sharp$.

Note that the above includes both over-approximating and under-approximating operations. In section 4.1, we relax the requirements and obtain an analysis where subtraction is the only under-approximating operation.

For a statement C and $n \in \mathcal{D}$, the approximate negative side $N^\sharp(C, n)$, which over-approximates $N(C, \gamma(n))$, is (non-recursively) defined as follows:

$$N^\sharp(a, n) = pre{+}fail^\sharp(a, n)$$
$$N^\sharp(C_1\,;\, C_2, n) = N^\sharp\big(C_1, N^\sharp(C_2, n)\big)$$

$$N^\sharp\big(([\varphi];C_1)+([\psi];C_2),n\big) = [\varphi, N^\sharp(C_1,n)]^\sharp \sqcup [\psi, N^\sharp(C_2,n)]^\sharp$$

$$N^\sharp\big(([\psi];C_{\text{body}})^*;[\varphi],n\big) = \text{the first } n_j \in \{n_i\}_{i\geq 0} \text{ such that } n_{j+1} \sqsubseteq n_j \text{ where}$$
$$n_0 = [\varphi,n]^\sharp \text{ and } n_{i+1} = n_i \triangledown [\psi, N^\sharp(C_{\text{body}}, n_i)]^\sharp$$

For a statement C and a pair of elements $p, n \in \mathcal{D}$ that are disjoint ($\gamma(p) \cap \gamma(n) = \varnothing$), we define the approximate positive side $P^\flat(C, p, n)$ such that it under-approximates $P(C, \mathcal{U} \smallsetminus \gamma(n))$. $P^\flat(C, p, n)$ is defined mutually with an auxiliary $Q^\natural(C, p, n)$ by induction on the structure of C. Optimally, $Q^\natural(C, p, n)$ represents a tight under-approximation of $P(C, \gamma(p))$, but actually need not be an under-approximation. Also, note how n is used to abstractly represent the complement of the set of interest.

For loop-free code, $P^\flat$ and $Q^\natural$ are (non-recursively) defined as follows:

$$\begin{aligned}
P^\flat(C,p,n) &= Q^\natural(C,p,n) - N^\sharp(C,n)\\
Q^\natural(a,p,n) &= wp^\flat(a,p)\\
Q^\natural(C_1\,;C_2,p,n) &= P^\flat\big(C_1, P^\flat(C_2,p,n), N^\sharp(C_2,n)\big)\\
Q^\natural\big(([\varphi];C_1)+([\psi];C_2),p,n\big) &= \big(P^\flat(C_1,p,n) \sqcap P^\flat(C_2,p,n)\big) \sqcup\\
&\qquad [\neg\psi, P^\flat(C_1,p,n)]^\flat \sqcup [\neg\varphi, P^\flat(C_2,p,n)]^\flat
\end{aligned}$$

For a loop $C_{\text{loop}} = ([\psi];C_{\text{body}})^*;[\varphi]$, we define a sequence $\{q_i\}_{i\geq 0}$ of approximants to $Q^\natural(C_{\text{loop}}, p, n)$, where $q_{i+1} = q_i \triangledown \tau(q_i)$ and the initial point q_0 and the transformer τ are defined following either the characterization (4) using an approximation $R^\natural_\exists \in \mathcal{D}$ of an existential recurrent set of the loop:

$$\begin{aligned}
q_0 &= \big(R^\natural_\exists - N^\sharp(C_{\text{loop}},n)\big) \sqcup [\neg\psi, p]^\flat\\
\tau(q_i) &= \big([\psi,p]^\flat \sqcap wp^\flat(C_{\text{body}}, q_i)\big) \sqcup [\neg\varphi, wp^\flat(C_{\text{body}}, q_i)]^\flat
\end{aligned}$$

or the characterization (5) using an approximation $R^\natural_\forall \in \mathcal{D}$ of a universal recurrent set:

$$\begin{aligned}
q_0 &= R^\natural_\forall \sqcup \big([\varphi,p]^\flat - N^\sharp(C_{\text{loop}},n)\big)\\
\tau(q_i) &= \big([\neg\varphi, pre^\sharp(C_{\text{body}}, q_i)]^\flat - N^\sharp(C_{\text{loop}},n)\big)
\end{aligned}$$

As for loop-free commands, $Q^\natural$ can be computed first, and $P^\flat$ defined using the result. That is, define $Q^\natural(C_{\text{loop}}, p, n) = q_j$ where p_j is the first element such that $q_{j+1} \sqsubseteq q_j$, and then define $P^\flat(C_{\text{loop}}, p, n) = Q^\natural(C_{\text{loop}}, p, n) - N^\sharp(C_{\text{loop}}, n)$.

Alternatively, $P^\flat$ and $Q^\natural$ can be computed simultaneously by also defining a sequence $\{p_i\}_{i\geq 0}$ of safe under-approximants of $P^\flat(C_{\text{loop}}, p, n)$, where $p_0 = q_0$ and $p_{i+1} = (p_i \triangledown \tau(q_i)) - N^\sharp(C_{\text{loop}}, n)$. Then $P^\flat(C_{\text{loop}}, p, n) = p_j$ where p_j is the first element such that $q_{j+1} \sqsubseteq q_j$ or $p_{j+1} \not\sqsupseteq p_j$. In this case, we may obtain a sound $P^\flat$ before the auxiliary $Q^\natural$ has stabilized. While we have not yet done rigorous experimental validation, we prefer this approach when dealing with coarse subtraction.

When analyzing a top-level program C_{prg}, the analysis starts with $N^\sharp(C_{\text{prg}}, \bot)$ and precomputes $N^\sharp$ (an over-approximation of unsafe states) for all statements of the program. Then it proceeds to compute $P^\flat(C_{\text{prg}}, \top, \bot)$ (an under-approximation of safe input states) reusing the precomputed results for $N^\sharp$.

Note that we are using join and widening on the positive side which means that $Q^\natural$ may not under-approximate the positive side of the concrete characterization. The use of widening allows for the ascending chain to converge, and subtraction of the negative side ensures soundness of $P^\flat$. In other words, while the alternate concrete characterizations (4) and (5) are used to guide the definition of the approximate characterizations, soundness is argued directly rather than by using (4) and (5) as an intermediate step.

Theorem 2. *For a statement C and $p, n \in \mathcal{D}$ s.t. $\gamma(p) \cap \gamma(n) = \varnothing$, $N^\sharp(C, n) \supseteq N(C, \gamma(n))$ and $P^\flat(C, p, n) \subseteq P(C, \mathcal{U} \smallsetminus \gamma(n))$. Hence, for a top-level program C_{prg}, $\gamma(N^\sharp(C_{\text{prg}}, \bot)) \supseteq N(C_{\text{prg}}, \varnothing)$ (i.e., it over-approximates input states that may lead to failure), and $\gamma(P^\flat(C_{\text{prg}}, \top, \bot)) \subseteq P(C_{\text{prg}}, \mathcal{U})$ (i.e., it under-approximates safe input states).*

The argument for $N^\sharp$ proceeds in a standard way [11]. Soundness for $P^\flat$ then follows due to the use of subtraction.

4.1 Optimizations of Constraints

Use of over-approximate operations Since we are subtracting $N^\sharp(C, n)$ anyway, we can relax the right-hand side of the definition of $Q^\natural(C, p, n)$ without losing soundness. Specifically, we can replace under-approximating and must- operations by their over-approximating and may- counterparts. This way, we obtain an analysis where subtraction is the only under-approximating operation.

- For a loop-free statement C, always use $pre^\sharp(C, p)$ in place of $wp^\flat(C, p)$ (note that we *already* use $pre^\sharp$ on the positive side for loop bodies when starting from a universal recurrent set). This can be handy, e.g., for power set domains where $pre^\sharp$ (unlike $wp^\flat$) can be applied element-wise. Also, these transformers may coincide for deterministic loop-free statements (if the abstraction is precise enough). Later, when discussing Example 2, we note some implications of this substitution.
- For a state formula φ, use $[\varphi, \cdot]^\sharp$ in place of $[\varphi, \cdot]^\flat$. Actually, for some combinations of an abstract domain and a language of formulas, these transformers coincide. For example, in a polyhedral domain, conjunctions of linear constraints with non-strict inequalities have precise representations as domain elements.
- For branching statements, use $[\varphi, P^\flat(C_1, p, n)]^\sharp \sqcup [\psi, P^\flat(C_2, p, n)]^\sharp$ in place of the original expression.
- In the definition of $Q^\natural$, an over-approximate meet operation $\sqcap^\sharp$ suffices.

The result of these relaxations is:

$$\begin{aligned} Q^\natural(a, p, n) &= pre^\sharp(a, p) \\ Q^\natural(C_1 \,;\, C_2, p, n) &= P^\flat\big(C_1, P^\flat(C_2, p, n), N^\sharp(C_2, n)\big) \\ Q^\natural\big(([\varphi]; C_1) + ([\psi]; C_2), p, n\big) &= [\varphi, P^\flat(C_1, p, n)]^\sharp \sqcup [\psi, P^\flat(C_2, p, n)]^\sharp \end{aligned}$$

$$\begin{aligned} q_0 &= \big(R^\natural_\exists - N^\sharp(C_{\text{loop}}, n)\big) \sqcup [\neg\psi, p]^\sharp \\ \tau(q_i) &= \big([\psi, p]^\sharp \sqcap^\sharp pre^\sharp(C_{\text{body}}, q_i)\big) \sqcup [\neg\varphi, pre^\sharp(C_{\text{body}}, q_i)]^\sharp \end{aligned}$$

or

$$q_0 = R^{\natural}_{\forall} \sqcup \left([\varphi, p]^{\sharp} - N^{\sharp}(C_{\text{loop}}, n)\right)$$
$$\tau(q_i) = \left([\neg\varphi, pre^{\sharp}(C_{\text{body}}, q_i)]^{\sharp} - N^{\sharp}(C_{\text{loop}}, n)\right)$$

No subtraction for $\mathbf{Q}^{\natural}$ For a similar reason, subtraction can be removed from the characterization of $Q^{\natural}$ without affecting soundness of $P^{\flat}$.

Bound on the positive side Another observation is that for a loop C_{loop} as in (2), the positive side $P(C_{\text{loop}}, S)$ is bounded by $[\![\neg\varphi]\!] \cup S$, as can be seen from the characterization (3). This can be incorporated into a specialized definition for loops, defining $P^{\flat}(C_{\text{loop}}, p, n) = \left(Q^{\natural}(C_{\text{loop}}, p, n) \sqcap ([\neg\varphi]^{\sharp} \sqcup p)\right) - N^{\sharp}(C_{\text{loop}}, n)$ or by performing the meet during computation of $Q^{\natural}$ by defining $q_{i+1} = \left(q_i \triangledown \tau(q_i)\right) \sqcap ([\neg\varphi]^{\sharp} \sqcup p)$.

4.2 Approximating the Recurrent Set

When approximating the positive side for a loop, the computation is initialized with an approximation of the recurrent set induced by the loop. Our analysis is able to start with either an existential or a universal recurrent set depending on what search procedure is available for the domain. The instantiation of our approach for numerical domains uses the tool E-HSF [5] that is capable of approximating both existential and universal recurrence. Other tools for numeric domains are described in [13,24]. The instantiation of our approach for the shape analysis with 3-valued logic uses a prototype procedure that we have developed to approximate existential recurrent sets.

Normally, the search procedures are incomplete: the returned sets only imply recurrence, and the search itself might not terminate (we assume the use of timeouts in this case). For this reason, in Sect. 3, we prefer not to define the recurrent sets to be maximal. This incompleteness leaves room for our analysis to improve the approximation of recurrence. For example, sometimes the solver produces a universal recurrence that is closed under forward propagation, but is not closed under backward propagation. In such cases, our analysis can produce a larger recurrent set.

5 Examples

In this section, we demonstrate our approach on several examples: first for a numeric domain, and then for the shape analysis domain of 3-valued structures. We note that numeric programs are considered here solely for the purpose of clarity of explanation, since the domain is likely to be familiar to most readers. We do not claim novel results specifically for the analysis of numeric programs, although we note that our approach may be able to complement existing tools. Detailed explanations of Examples are included in the companion technical report [3].

Example 1 aims at describing steps of the analysis in detail (to the extent allowed by space constraints). Example 2 is restricted to highlights of the analysis and includes a pragmatic discussion on using $pre^{\sharp}$ on the positive side. Examples 3 and 4 consider programs from a shape analysis domain and we only report on the result of the analysis.

```
1  while x ≥ 1 do
2    if x = 60 then
3      x ← 50
4    end
5    x ← x + 1
6    if x = 100 then
7      x ← 0
8    end
9  end
10 assert 0
```

(a) With syntactic sugar.

$$
\begin{aligned}
&{}_1\big([x \geq 1];\\
&\quad {}_2(([x = 60]; {}_3x \leftarrow 50) + ([x \neq 60]; \mathtt{skip}));\\
&\quad {}_5x \leftarrow x + 1;\\
&\quad {}_6(([x = 100]; {}_7x \leftarrow 0) + ([x \neq 100]; \mathtt{skip}));\\
&\big)^*; [x \leq 0];\\
&{}_{10}\,\mathtt{assert}\,0
\end{aligned}
$$

(b) Desugared.

Fig. 2: Example program 1.

Example 1. In this example, we consider the program in Fig. 2: Fig. 2a shows program text using syntactic sugar for familiar while-language, and Fig. 2b shows the corresponding desugared program. We label the statements that are important for the analysis with the corresponding line numbers from Fig. 2a (like in ${}_3x \leftarrow 50$).

We assume that program variables (just x in this case) take *integer* values. For the abstract domain, we use disjunctive refinement over intervals allowing a bounded number of disjuncts (e.g., [2]). Recall that $\langle x : [a; b], y : [c; d]\rangle$ denotes a singleton abstract state of a program with two variables x and y, representing the set of concrete states, satisfying $(a \leq x \leq b) \wedge (c \leq y \leq d)$. Note that for this abstract domain and the formulas, appearing in the program, $[\cdot]^\flat$ and $[\cdot]^\sharp$ coincide, and we write $[\cdot]^\natural$ to denote either. To emphasize that the analysis can produce useful results even when using a coarse subtraction function, we use subtraction as defined in (1). That is, we just drop from the positive side those disjuncts that have a non-empty intersection with the negative side. For example, $\{\langle x : [1; 3]\rangle, \langle x : [5; 7]\rangle\} - \langle x : [6; 8]\rangle = \langle x : [1; 3]\rangle$. The analysis is performed mechanically by a prototype tool that we have implemented.

To simplify the presentation, in this example, we bound the number of disjuncts in a domain element by 2. Also to simplify the presentation, we omit the $\sharp$- and $\flat$-superscripts, and write, e.g., *pre*+*fail* for *pre*+*fail*$^\sharp$. For a statement labeled with i, we write N_i^j to denote the result of the j-th step of the computation of its negative side, and N_i to denote the computed value (similarly, for P).

We start with the analysis of the negative side. For the final statement,

$$N_{10}^1 = \textit{pre+fail}(\mathtt{assert}\,0, \bot) = \top$$

then, we proceed to the first approximation for the loop (for clarity, we compute *pre* of the body in steps),

$$
\begin{aligned}
N_1^1 &= [x \leq 0, N_{10}^1]^\natural = \langle x : (-\infty; 0]\rangle\\
N_7^1 &= \textit{pre+fail}(x \leftarrow 0, N_1^1) = \top\\
N_6^1 &= [x = 100, N_7^1]^\natural \sqcup [x \neq 100, N_1^1]^\natural = \{\langle x : (-\infty; 0]\rangle, \langle x : [100]\rangle\}\\
N_5^1 &= \textit{pre+fail}(x \leftarrow x + 1, N_6^1) = \{\langle x : (-\infty; -1]\rangle, \langle x : [99]\rangle\}
\end{aligned}
$$

$$N_3^1 = \mathit{pre}\text{+}\mathit{fail}(x \leftarrow 50, N_5^1) = \bot$$
$$N_2^1 = [x = 60, N_3^1]^\sharp \sqcup [x \neq 60, N_5^1]^\sharp = \{\langle x : (-\infty; -1]\rangle, \langle x : [99]\rangle\}$$
$$N_1^2 = N_1^1 \sqcup [x \geq 1, N_2^1]^\sharp = \{\langle x : (-\infty; 0]\rangle, \langle x : [99]\rangle\}$$

then, repeating the same similar sequence of steps for the second time gives

$$N_1^2 = N_1^2 \sqcup [x \geq 1, N_2^2]^\sharp = \{\langle x : (-\infty; 0]\rangle, \langle x : [98, 99]\rangle\}$$

at which point we detect an unstable bound. The choice of widening strategy is not our focus here, and for demonstration purposes, we proceed without widening, which allows to discover the stable bound of 61. In a real-world tool, to retain precision, some form of widening *up to* [14] or landmarks [23] could be used. Thus, we take

$$N_1 = \{\langle x : (-\infty; 0]\rangle, \langle x : [61; 99]\rangle\}$$
$$N_2 = \{\langle x : (-\infty; -1]\rangle, \langle x : [61; 99]\rangle\} \qquad N_6 = \{\langle x : (-\infty; 0]\rangle, \langle x : [61; 100]\rangle\}$$
$$N_3 = \bot \qquad N_7 = \top$$
$$N_5 = \{\langle x : (-\infty; -1]\rangle, \langle x : [61; 99]\rangle\} \qquad N_{10} = \top$$

To initialize the positive side for the loop, we use a universal recurrent set obtained by three calls to E-HSF with different recurrent set templates. The result is $R_\forall = \{\langle x : [4; 60]\rangle, \langle x : [100; +\infty)\rangle\}$. Note that in this example, universal recurrence and safety coincide, and our analysis will be able to improve the result by showing that the states in $\langle x : [1; 3]\rangle$ are also safe. Since we are using a power set domain, we choose to use *pre* instead of *wp* for the final statement (as described in Sect. 4.1), not just for the loop (where we need to use it due to starting with $R_\forall$). We start with

$$P_{10}^1 = \mathit{pre}(\texttt{assert}\,0, \top) - N_{10} = \bot - N_{10} = \bot$$

then proceed to the loop (again, computing *pre* of its body in steps),

$$P_1^1 = R_\forall \sqcup [x \leq 0, P_{10}^1]^\sharp - N_1 = \{\langle x : [4; 60]\rangle, \langle x : [100; +\infty)\rangle\}$$
$$P_7^1 = \mathit{pre}(x \leftarrow 0, P_1^1) - N_7 = \bot$$
$$\begin{aligned} P_6^1 &= [x = 100, P_7^1]^\sharp \sqcup [x \neq 100, P_1^1]^\sharp - N_6 \\ &= \{\langle x : [4 : 60]\rangle, \langle x : [101; +\infty)\rangle\} - N_6 \\ &= \{\langle x : [4 : 60]\rangle, \langle x : [101; +\infty)\rangle\} \end{aligned}$$
$$P_5^1 = \mathit{pre}(x \leftarrow x + 1, P_6^1) - N_5 = \{\langle x : [3 : 59]\rangle, \langle x : [100; +\infty)\rangle\}$$
$$P_3^1 = \mathit{pre}(x \leftarrow 50, P_5^1) - N_3 = \top$$
$$\begin{aligned} P_2^1 &= [x = 60, P_3^1]^\sharp \sqcup [x \neq 60, P_5^1]^\sharp - N_2 \\ &= \{\langle x : [3; 59]\rangle, \langle x : [60]\rangle, \langle x : [100; +\infty)\rangle\} - N_2 \\ &= \{\langle x : [3; 60]\rangle, \langle x : [100; +\infty)\rangle\} \end{aligned}$$
$$P_1^2 = (P_1^1 \sqcup ([x \geq 1, P_2^1]^\sharp - N_2)) - N_2 = \{\langle x : [3; 60]\rangle, \langle x : [100; +\infty)\rangle\}$$

at which point we detect an unstable bound, but we again proceed without widening and are able to discover the stable bound of 1. Also note that (as observed in Sect. 4.1),

P_1 is bounded by $P_{10} \sqcup [\neg x \leq 0]^\sharp = \langle x : [1; +\infty)\rangle$. This bound could be used to improve the result of widening. Thus, we take

$$
\begin{array}{ll}
P_1 = \{\langle x : [1; 60]\rangle, \langle x : [100; +\infty)]\rangle\} & \\
P_2 = \{\langle x : [0; 60]\rangle, \langle x : [100; +\infty)]\rangle\} & P_6 = \{\langle x : [1; 60]\rangle, \langle x : [101; +\infty)]\rangle\} \\
P_3 = \top & P_7 = \bot \\
P_5 = \{\langle x : [0; 59]\rangle, \langle x : [100; +\infty)]\rangle\} & P_{10} = \bot
\end{array}
$$

Thus, in this example, our analysis was able to prove that initial states $\{\langle x : [1; 60]\rangle, \langle x : [100; +\infty)]\rangle\}$ are safe, which is a slight improvement over the output of E-HSF.

Example 2. In this example, we consider the program in Fig. 3. In the program, $*$ stands for a value non-deterministically chosen at runtime. All the assumptions made for Example 1 are in effect for this one as well, except that we increase the bound on the size of the domain element to 4. The analysis is able to produce the following approximation of the safe entry states:

$$
\begin{array}{c}
\{\langle x : [100; +\infty), y : \top\rangle, \langle x : (-\infty; 0], y : (-\infty; -1]\rangle, \\
\langle x : (-\infty; 0], y : [1; +\infty)\rangle, \langle x : [1; 99], y : [1; +\infty)\rangle\}
\end{array}
$$

This example also displays an interplay between coarse subtraction and the use of over-approximate operations (especially, *pre*) on the positive side. In order to retain precision when coarse subtraction is used, it seems important to be able to keep the positive side partitioned into a number of disjuncts. In a real-world analysis, this can be achieved, e.g., by some form of trace partitioning [16]. In this example, we employ a few simple tricks, one of those can be seen from Fig. 3. Observe that we translated the non-deterministic condition in lines 3-7 of the syntactically sugared program (Fig. 3a) into equivalent nested conditions (statement 3 of the desugared program in Fig. 3b) which allows the necessary disjuncts to emerge on the positive side.

Shape Analysis Examples In what follows, we demonstrate our approach for a shape analysis domain. We treat two simple examples using the domain of 3-valued structures, and we claim that our approach provides a viable decomposition of backward analysis (for this domain and probably for some other shape analysis domains). For background information on shape analysis with 3-valued logic, please refer to [22] and accompanying papers, e.g., [19,1,25]. To handle the examples, we use a mechanized procedure built on top of the TVLA shape analysis tool (`http://www.cs.tau.ac.il/~tvla/`).

Example 3. In this example, we consider the program in Fig. 4. The program manipulates a pointer variable x, and the heap cells each have a pointer field n. We compare x to *nil* to check whether it points to a heap cell. We write $x \to n$ to denote access to the pointer field n of the heap cell pointed to by x. The program in Fig. 4 just traverses its input structure in a loop.

The analysis identifies that both cyclic and acyclic lists are safe inputs for the program – and summarizes them in eight and nine shapes respectively. Figures 6 and 7 show examples of the resulting shapes.

```
1  while x ≥ 1 do
2    if x ≤ 99 then
3      if y ≤ 0 ∧ ∗ then
4
5        assert 0
6
7      end
8      if ∗ then
9        x ← −1
10     end
11   end
12   x ← x + 1
13 end
14 assert y ≠ 0
```

(a) With syntactic sugar.

$$
\begin{aligned}
&{}_1([x \geq 1];\\
&\quad {}_2(([x \leq 99];\\
&\qquad {}_3(([y \leq 0]; {}_4({}_5\,\texttt{assert}\,0 + \texttt{skip}))\\
&\qquad\quad + ([y \geq 1]; \texttt{skip}));\\
&\qquad {}_8({}_9 x \leftarrow -1 + \texttt{skip})\\
&\quad) + ([x \geq 100]; \texttt{skip}));\\
&\quad {}_{12} x \leftarrow x + 1\\
&)^*; [x \leq 0];\\
&{}_{14}\,\texttt{assert}\; y \neq 0
\end{aligned}
$$

(b) Desugared.

Fig. 3: Example program 2.

```
1  while x ≠ nil do
2    x ← (x → n)
3  end
```

Fig. 4: Example program 3.

```
1  while x ≠ nil do
2    x ← (x → n)
3    x ← (x → n)
4  end
```

Fig. 5: Example program 4.

Example 4. In this example, we consider the program in Fig. 5. In this program, the loop body makes two steps through the list instead of just one. While the first step (at line 2) is still guarded by the loop condition, the second step (at line 3) is a source of failure. That is, the program fails when given a list of odd length as an input. The abstraction that we employ is not expressive enough to encode such constraints on the length of the list. The analysis is able to show that cyclic lists represent safe inputs, but the only acyclic list that the analysis identifies as safe is the list of length exactly two.

6 Related Work

In [15], a backward shape analysis with 3-valued logic is presented that relies on the correspondence between 3-valued structures and first-order formulas [25]. It finds an over-approximation of states that may lead to failure, and then (as 3-valued structures do not readily support complementation) the structures are translated to an equivalent quantified first-order formula, which is then negated. This corresponds to approximating the negative side in our approach and then taking the complement, with the exception that the result is not represented as an element of the abstract domain. At least in principle, the symbolic abstraction $\hat{\alpha}$ of [20] could map back to the abstract domain.

For shape analysis with separation logic [21], preconditions can be inferred using a form of abduction called bi-abduction [6]. The analysis uses an over-approximate

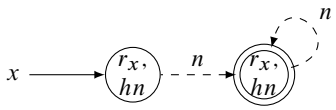


Fig. 6: Example of a safe structure causing non-termination.

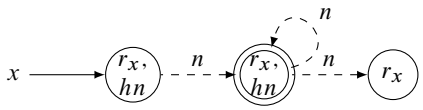


Fig. 7: Example of a safe structure leading to successful termination.

abstraction, and it includes a filtering step that checks generated preconditions (by computing their respective postconditions) and discards the unsound ones. The purpose of the filtering step – keeping soundness of a precondition produced with over-approximate abstraction – is similar to our use of the negative side.

For numeric programs, the problem of finding preconditions for safety has seen some attention lately. In [18], a numeric analysis is presented that is based primarily on over-approximation. It simultaneously computes the representations of two sets: of states that may lead to successful termination, and of states that may lead to failure. Then, meet and generic negation are used to produce representations of states that cannot fail, states that must fail, etc. An under-approximating backward analysis for the polyhedral domain is presented in [17]. The analysis defines the appropriate under-approximate abstract transformers and to ensure termination, proposes a lower widening based on the generator representation of polyhedra. With E-HSF [5], the search for preconditions can be formulated as solving $\forall\exists$ quantified Horn clauses extended with well-foundedness conditions. The analysis is targeted specifically at linear programs, and is backed by a form of counterexample-guided abstraction refinement.

7 Conclusion and Future Work

We presented an alternative decomposition of backward analysis, suitable for domains that do not readily support complementation and under-approximation of greatest fixed points. Our approach relies on an under-approximating subtraction operation and a procedure that finds recurrent sets for loops – and builds a sequence of successive under-approximations of the safe states. This decomposition allowed us to implement a backwards analysis for the domain of 3-valued structures and to obtain acceptable analysis results for two simple programs.

For shape analysis examples, we employed quite a simplistic procedure to approximate a recurrent set. One direction for future research is into recurrence search procedures for shape analysis that are applicable to realistic programs.

Another possible direction is to explore the settings where non-termination counts as failure. This is the case, e.g., when checking abstract counterexamples for concrete feasibility [4].

Acknowledgements We thank Andrey Rybalchenko for helpful discussion and assistance with E-HSF, and Mooly Sagiv and Roman Manevich for sharing the source code of TVLA. A. Bakhirkin is supported by a Microsoft Research PhD Scholarship.

References

1. Arnold, G., Manevich, R., Sagiv, M., Shaham, R.: Combining shape analyses by intersecting abstractions. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI. LNCS, vol. 3855, pp. 33–48. Springer (2006)
2. Bagnara, R., Hill, P.M., Zaffanella, E.: Widening operators for powerset domains. STTT 9(3-4), 413–414 (2007)
3. Bakhirkin, A., Berdine, J., Piterman, N.: Backward analysis via over-approximate abstraction and under-approximate subtraction. Tech. Rep. MSR-TR-2014-82, Microsoft Research (2014)
4. Berdine, J., Bjørner, N., Ishtiaq, S., Kriener, J.E., Wintersteiger, C.M.: Resourceful reachability as HORN-LA. In: McMillan, K.L., Middeldorp, A., Voronkov, A. (eds.) LPAR. LNCS, vol. 8312, pp. 137–146. Springer (2013)
5. Beyene, T.A., Popeea, C., Rybalchenko, A.: Solving existentially quantified Horn clauses. In: Sharygina, N., Veith, H. (eds.) CAV. LNCS, vol. 8044, pp. 869–882. Springer (2013)
6. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. In: Shao, Z., Pierce, B.C. (eds.) POPL. pp. 289–300. ACM (2009)
7. Calcagno, C., Ishtiaq, S.S., O'Hearn, P.W.: Semantic analysis of pointer aliasing, allocation and disposal in Hoare logic. In: PPDP. pp. 190–201 (2000)
8. Calcagno, C., Yang, H., O'Hearn, P.W.: Computability and complexity results for a spatial assertion language for data structures. In: APLAS. pp. 289–300 (2001)
9. Clarke, E.M.: Program invariants as fixed points (preliminary reports). In: FOCS. pp. 18–29. IEEE Computer Society (1977)
10. Cousot, P.: Semantic foundations of program analysis. In: Muchnick, S.S., Jones, N.D. (eds.) Program Flow Analysis: Theory and Applications, pp. 303–342. Prentice-Hall (1981)
11. Cousot, P., Cousot, R.: Abstract interpretation and application to logic programs. J. Log. Program. 13(2&3), 103–179 (1992)
12. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Aho, A.V., Zilles, S.N., Szymanski, T.G. (eds.) POPL. pp. 84–96. ACM Press (1978)
13. Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.G.: Proving non-termination. In: Necula, G.C., Wadler, P. (eds.) POPL. pp. 147–158. ACM (2008)
14. Halbwachs, N., Proy, Y.E., Roumanoff, P.: Verification of real-time systems using linear relation analysis. Form. Method. Syst. Des. 11(2), 157–185 (1997)
15. Lev-Ami, T., Sagiv, M., Reps, T., Gulwani, S.: Backward analysis for inferring quantified preconditions. Tech. Rep. TR-2007-12-01, Tel Aviv University (Dec 2007)
16. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: Sagiv, S. (ed.) ESOP. LNCS, vol. 3444, pp. 5–20. Springer (2005)
17. Miné, A.: Inferring sufficient conditions with backward polyhedral under-approximations. Electr. Notes Theor. Comput. Sci. 287, 89–100 (2012)
18. Popeea, C., Chin, W.N.: Dual analysis for proving safety and finding bugs. Sci. Comput. Program. 78(4), 390–411 (2013)
19. Reps, T.W., Sagiv, S., Loginov, A.: Finite differencing of logical formulas for static analysis. In: Degano, P. (ed.) ESOP. LNCS, vol. 2618, pp. 380–398. Springer (2003)
20. Reps, T.W., Sagiv, S., Yorsh, G.: Symbolic implementation of the best transformer. In: Steffen, B., Levi, G. (eds.) VMCAI. LNCS, vol. 2937, pp. 252–266. Springer (2004)
21. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS. pp. 55–74. IEEE Computer Society (2002)
22. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24(3), 217–298 (2002)

23. Simon, A., King, A.: Widening polyhedra with landmarks. In: Kobayashi, N. (ed.) APLAS. LNCS, vol. 4279, pp. 166–182. Springer (2006)
24. Velroyen, H., Rümmer, P.: Non-termination checking for imperative programs. In: Beckert, B., Hähnle, R. (eds.) TAP. LNCS, vol. 4966, pp. 154–170. Springer (2008)
25. Yorsh, G., Reps, T.W., Sagiv, M., Wilhelm, R.: Logical characterizations of heap abstractions. ACM Trans. Comput. Log. 8(1) (2007)