An Algorithm Inspired by Constraint Solvers to Infer Inductive Invariants in Numeric Programs*

Antoine Miné[1], Jason Breck[2], and Thomas Reps[2,3]

[1] Sorbonne Universités, UPMC Univ Paris 06, CNRS, LIP6, Paris, France
[2] University of Wisconsin; Madison, WI, USA
[3] GrammaTech, Inc.; Ithaca, NY, USA

Abstract. This paper addresses the problem of proving a given invariance property φ of a loop in a numeric program, by inferring automatically a stronger inductive invariant ψ. The algorithm we present is based on both *abstract interpretation* and *constraint solving*. As in abstract interpretation, it computes the effect of a loop using a numeric abstract domain. As in constraint satisfaction, it works from "above"—interactively splitting and tightening a collection of abstract elements until an inductive invariant is found. Our experiments show that the algorithm can find non-linear inductive invariants that cannot normally be obtained using intervals (or octagons), even when classic techniques for increasing abstract-interpretation precision are employed—such as increasing and decreasing iterations with extrapolation, partitioning, and disjunctive completion. The advantage of our work is that because the algorithm uses standard abstract domains, it sidesteps the need to develop complex, non-standard domains specialized for solving a particular problem.

1 Introduction

A key concept in formal verification of programs is that of invariants, i.e., properties true of all executions of a program. For instance, safety properties, such as the fact that program variables stay within their expected bounds, are invariance properties, while more complex properties, such as termination, often depend crucially on invariants. Hence, a large part of program-verification research concerns methods for checking or inferring suitable invariants. Invariants are particularly important for loops: an invariant for a loop provides a single

* The authors wish to thank Sriram Sankaranarayanan and Charlotte Truchet for the very useful discussions that inspired this work. The work was supported, in part, by the project ANR-15-CE25-0002 Coverif from the French Agence Nationale de la Recherche; by a gift from Rajiv and Ritu Batra; by DARPA under cooperative agreement HR0011-12-2-0012; by AFRL under DARPA MUSE award FA8750-14-2-0270 and DARPA STAC award FA8750-15-C-0082; and by the UW-Madison Office of the Vice Chancellor for Research and Graduate Education with funding from the Wisconsin Alumni Research Foundation. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring agencies.

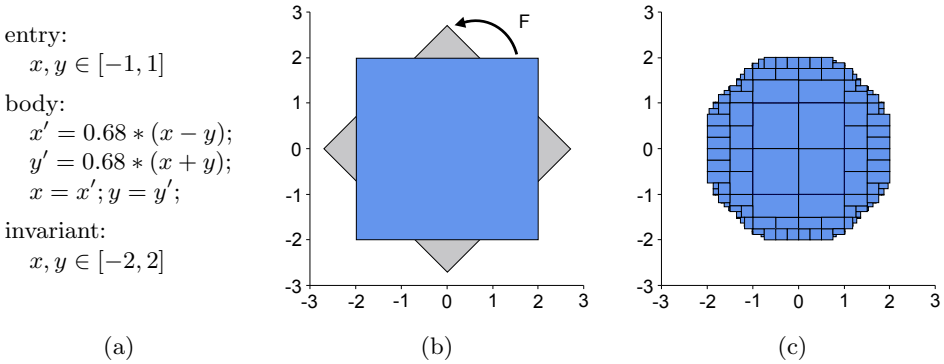


Fig. 1. (a) A loop that performs a 45-degree rotation with a slight inward scaling; (b) no box is an inductive invariant; (c) inductive invariant obtained by our algorithm.

property that holds on every loop iteration, and relieves one from having to prove the safety of each loop iteration individually (which might not be possible for infinite-state systems when the number of loop iterations is unbounded, and not practical when it is finite but large). The main principle for proving that a property is an invariant for a loop is to find a stronger property that is an *inductive invariant*, i.e., it holds when entering the loop for the first time, and is stable across a single loop iteration.

While an invariant associated with a safety property of interest can be relatively simple (e.g., a variable's value is bounded by such-and-such a quantity), it may not always be inductive. In general, it is a much more complex task to find an inductive invariant: inductive invariants can have much more complex shapes, as shown by the following example.

Consider the program in Fig. 1. It has two variables, x and y, and its loop body performs a 45-degree rotation of the point (x, y) about the origin, with a slight inward scaling. Additionally, we are given a precondition telling us that, upon entry to the loop, $x \in [-1, 1]$ and $y \in [-1, 1]$. Intuitively, no matter how many times we go around this loop, the point (x, y) will not move far away from the origin. To make this statement precise, we choose a bounding box that is aligned with the axes, such as $I \stackrel{\text{def}}{=} [-2, 2] \times [-2, 2]$, and observe that the program state, considered as a point (x, y), is guaranteed to lie inside I every time that execution reaches the head of the loop. Consequently, I is an invariant at the head of the loop.

Even though I is an invariant at the head of the loop, *it is not an inductive invariant*, because there are some points in I, such as its corners, that do not remain inside I after a 45-degree rotation. Actually, the corner points are not reachable states; however, there is no way to express that fact using an axis-aligned box. In fact, no box is an inductive invariant for this program.

To express symbolically the distinction between invariants and inductive invariants, we write $E = [-1, 1] \times [-1, 1]$ for the set of states that satisfy the precondition, and write $F(X) \stackrel{\text{def}}{=} \{(0.68 * (x - y), 0.68 * (x + y)) \mid (x, y) \in X\}$ for the function that represents the transformation performed by the loop body. The

set of program states reachable at the loop head after any number of iterations is then $\bigcup_{n\in\mathbb{N}} F^n(E)$. The box I is an invariant at the loop head because I includes all states reachable at the loop head; that is, $\bigcup_{n\in\mathbb{N}} F^n(E) \subseteq I$. However, I is not an inductive invariant because $F(I) \not\subseteq I$.

Suppose that we wish to prove that I is an invariant. It is not practical to do so by directly computing the set of reachable states and checking whether that is a subset of I, because that would require executing the loop an extremely large (and possibly infinite) number of times. Instead, we will look for a set of boxes like J, shown in Fig. 1(c). One iteration of the loop maps this set of boxes into a subset of itself: the inward scaling, along with the rotation, maps the entire shape, including the jagged edges, into its own interior. Thus, even though no single box is an inductive invariant, a set of boxes, such as J, can be an inductive invariant. Note that this property can be verified by analyzing the effect of just one iteration of the loop applied to J. However, as this example illustrates, an inductive invariant like J may be significantly more complicated, and therefore more difficult to find, than an invariant such as I.

This paper is concerned with automatically strengthening a given invariant into an inductive invariant for numeric programs that manipulate reals or floats. In contrast to classical fixpoint-finding methods that start from the bottom of a lattice of abstract values and iteratively work upwards (towards an overapproximation of the infinite union mentioned above), the algorithm described in Sec. 4 starts with the invariant that the user wishes to prove and incrementally strengthens it until an inductive invariant is found.

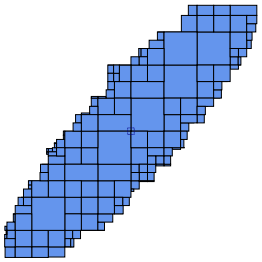
The contributions of our work include the following:

- We demonstrate that ideas and techniques from constraint programming, such as an abstract domain consisting of disjunctions of boxes, and operators for splitting and tightening boxes, can be used to find inductive invariants.
- We present an algorithm based on those techniques and demonstrate that it is able to find inductive invariants for numerical, single-loop programs that manipulate reals or floats.
- We demonstrate the generality of our approach by extending the algorithm to optionally use an abstract domain based on octagons instead of boxes.
- We further extend our algorithm to optionally run in a "relatively-complete mode", which guarantees that an inductive invariant will be found if one exists in the appropriate abstract domain and can be proven inductive using boxes.

Organization. Sec. 2 presents an overview of our method and its connections to abstract interpretation and constraint programming. Sec. 3 recalls key facts about invariants, abstract interpretation, and constraint programming. Sec. 4 presents a basic version of our algorithm, using boxes to simplify the presentation. Sec. 5 discusses extensions, including the use of other numeric domains, methods to tighten invariants, and a modified version of the algorithm that has a relative-completeness property. Sec. 6 discusses our prototype implementation and preliminary experiments. Sec. 7 surveys related work. Sec. 8 concludes.

entry:
$s_0, s_1 \in [-0.1, 0.1]$
body:
$r = 1.5 \times s_0 - 0.7 \times s_1 + [-0.1, 0.1];$
$s_1 = s_0;$
$s_0 = r;$
invariant:
$s_0, s_1 \in [-4, 4]$

(a)



(b)

Fig. 2. (a) Second-order digital-filter program, and (b) the inductive invariant found by our method, composed of 181 boxes.

2 Overview

Consider Fig. 2(a), which presents a model of a loop that implements a second-order digital filter. Its state is composed of two variables, s_0 and s_1, which are initially set to values in $[-0.1, 0.1]$. On each loop iteration, the value of s_0 is shifted into s_1, while s_0 is assigned $1.5 \times s_0 - 0.7 \times s_1 + [-0.1, 0.1]$ (using a temporary variable r). The evaluation of the interval $[-0.1, 0.1]$ is understood as choosing a fresh value between -0.1 and 0.1 on each loop iteration. The evaluation occurs in the real field, and the use of an interval allows modeling rounding errors that can occur in a floating-point implementation of the filter. Note that the program is nondeterministic due to the indeterminacy in the initial state and in the evaluation of the interval, with infinitely many executions and an infinite state space.

We wish to prove that, on each iteration, the property $s_0, s_1 \in [-4, 4]$ holds. This property is indeed invariant; however, it is not inductive—and no box is inductive—for reasons similar to those explained in Sec. 1. More generally, interval arithmetic [23] is ineffective at proving that box invariants hold. Control theory teaches us that there exist quadratic properties—ellipsoids—that can be inductive. Verification techniques that exploit this knowledge *a priori* can be developed, such as [12]. However, that approach constitutes a program-specific method. Fig. 2(b) shows an inductive invariant found by our method in 76 ms. While the shape of an ellipsoid is roughly recognizable, the inductive invariant found is actually the set union of 181 non-overlapping boxes. Our method employs only sets of boxes and does not use any knowledge from control theory.

Abstract interpretation. Abstract interpretation [8] provides tools to infer inductive invariants. It starts from the observation that the most-precise loop invariant is inductive and can be expressed mathematically as a least fixpoint or, constructively, as the limit of an iteration sequence. In general, the most-precise loop invariant is not computable because the iterates live in infinite-state spaces, and the iteration sequences can require a transfinite number of iterations to reach the least fixpoint. Abstract interpretation solves the first issue by reasoning in an abstract domain of simpler, computable properties (such as intervals [8] or octagons [20]), and the second issue by using extrapolation operators (widenings and narrowings). Its effectiveness relies on the choice of an appropriate abstract

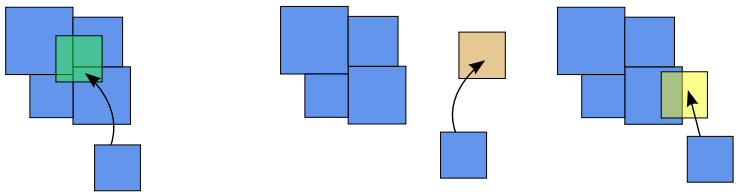


Fig. 3. Examples of boxes that should be (a) kept, (b) discarded, or (c) split, based on the intersection of each box's image with the other boxes.

domain, which must be able to represent inductive invariants, and also to support sufficiently precise operators—both to model program instructions and to perform extrapolation.

The Astrée analyzer [4] makes use of a large number of abstract domains, some of which are specific to particular application domains. In particular, the use of digital filters similar to Fig. 2(a) in aerospace software, and the inability of the interval domain to find any inductive invariant for them, motivated the design of the digital-filter domain [12] and its inclusion in Astrée. Note that Fig. 2(b) shows that an inductive invariant can, in fact, be expressed using a disjunction of boxes. However, applying classical disjunctive-domain constructions, such as disjunctive completion, or state- or trace-partitioning over the interval domain does not help in finding an inductive invariant. Indeed, these methods rely on program features, such as tests, control-flow joins, or user-provided hints, none of which occur in Fig. 2(a).

Constraint programming. Constraint programming [22] is a declarative programming paradigm in which problems are first expressed as conjunctions of first-order logic formulas, called constraints, and then solved by generic methods. Different families of constraints come with specific operators—such as choice operators and propagators—used by the solver to explore the search space of the problem and to reduce its size, respectively.

Algorithm synopsis. The algorithm presented in Sec. 4 was inspired by one class of constraint-solving algorithms—namely, continuous constraint solving—but applies those ideas to solve fixpoint equations. The algorithm works by iteratively refining a set of boxes until it becomes an inductive invariant. Fig. 3 illustrates the main rules used to refine the set of boxes. When the boxes form an inductive invariant, one iteration of the loop maps each box into the areas covered by other boxes (Fig. 3(a)). To refine a set of boxes that is not yet an inductive invariant, two refinement rules are applied:

- Boxes that map to areas outside of all other boxes are discarded (Fig. 3(b)).
- If a box maps to an area partially inside and partially outside the set, then we split it into two boxes (Fig. 3(c)).

These rules, along with a few others, are applied until either the set of boxes is inductive or failure is detected.

Even when an inductive invariant can be represented in a given abstract domain, approximation due to widening can prevent an abstract interpreter from

$$
\begin{array}{l}
\mathit{prog} ::= \textbf{assume } \mathit{entry} : \mathit{bexpr}; \\
\qquad \textbf{while true do} \\
\qquad\quad \textbf{assert } \mathit{inv} : \mathit{bexpr}; \\
\qquad\quad \mathit{body} : \mathit{stat} \\
\qquad \textbf{done}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{expr} ::= V & & V \in \mathcal{V} \\
\quad | \ [a, b] & & a, b \in \mathbb{R} \\
\quad | \ -\mathit{expr} & & \\
\quad | \ \mathit{expr} \circ \mathit{expr} & & \circ \in \{+, -, \times, /\}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{stat} ::= V \leftarrow \mathit{expr} & & V \in \mathcal{V} \\
\quad | \ \textbf{if } \mathit{bexpr} \textbf{ then } \mathit{stat} \textbf{ else } \mathit{stat} & & \\
\quad | \ \mathit{stat}; \mathit{stat} & &
\end{array}
$$

$$
\begin{array}{lll}
\mathit{bexpr} ::= \mathit{expr} \bowtie \mathit{expr} & & \bowtie \in \{<, \leq, =\} \\
\quad | \ \neg \mathit{bexpr} & & \\
\quad | \ \mathit{bexpr} \circ \mathit{bexpr} & & \circ \in \{\wedge, \vee\}
\end{array}
$$

Fig. 4. Simple programming language.

finding any inductive invariant—let alone the best one. The version of our algorithm presented in Sec. 4 shares this deficiency (although not because widening is involved); however, we modify our method to overcome this problem in Sec. 5.3.

The algorithm presented in this paper exploits simple, generic abstract domains (such as intervals) to compute effectively and efficiently the effect of a loop iteration, but (i) replaces extrapolation with a novel search algorithm inspired from constraint programming, and (ii) proposes a novel method to introduce disjunctions. Because of these features of the algorithm, it can sidestep the need to invent specialized abstract domains for specialized settings. For instance, a special digital-filter abstract domain (like the one created for Astrée [12]) is not needed to handle a digital-filter program; the interval domain suffices.

3 Terminology and Notation

3.1 Language and Semantics

The language we consider is defined in Figs. 4 and 5. It is a simple numeric language, featuring real-valued variables $V \in \mathcal{V}$; numeric expressions *expr*, including the usual operators and non-deterministic random choice ($[a, b]$); Boolean expressions *bexpr*; and various statements *stat*—assignments ($V \leftarrow \mathit{expr}$), conditionals (**if** *bexpr* **then** *stat* **else** *stat*) and sequences ($\mathit{stat}; \mathit{stat}$). A program *prog* consists of a single loop that starts in an entry state *entry* specified by a Boolean expression; a program is non-terminating, and executes a *body* statement. The loop invariant to prove (a Boolean expression) is explicitly present in the loop. Furthermore, the semantics is in terms of real numbers; if a floating-point semantics is desired instead, any rounding error has to be made explicit, using for instance the technique described in [19].

The concrete semantics of the language is presented in Fig. 5. It is a simple numeric, big-step, non-deterministic semantics, where expressions evaluate to a set of values $\mathbb{E}[\![e]\!]\rho \in \mathcal{P}(\mathbb{R})$ given an environment $\rho \in \mathcal{E} \stackrel{\text{def}}{=} \mathcal{V} \to \mathbb{R}$; Boolean expressions act as environment filters $\mathbb{B}[\![b]\!] : \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$; and statements as environment transformers $\mathbb{S}[\![s]\!] : \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$. Given a program whose entry states are given by $\mathit{entry} \in \mathit{bexpr}$, a formula for a candidate invariant for the loop $\mathit{inv} \in \mathit{bexpr}$, and loop body $\mathit{body} \in \mathit{stat}$, the goal is to prove that *inv* is indeed an invariant for the loop. This problem can be expressed as the following fixpoint problem:

$$\operatorname{lfp} \mathbb{P} \subseteq \mathbb{B}[\![\mathit{inv}]\!]\mathcal{E} \text{ where } \mathbb{P}(X) \stackrel{\text{def}}{=} \mathbb{B}[\![\mathit{entry}]\!]\,\mathcal{E} \cup \mathbb{S}[\![\mathit{body}]\!]X. \tag{1}$$

$$\mathbb{E}[\![\, expr \,]\!] : \mathcal{E} \to \mathcal{P}(\mathbb{R})$$

$$\mathbb{E}[\![\, V \,]\!]\rho \stackrel{\text{def}}{=} \{\, \rho(V) \,\}$$
$$\mathbb{E}[\![\, e_1 \circ e_2 \,]\!]\rho \stackrel{\text{def}}{=} \{\, v_1 \circ v_2 \mid v_1 \in \mathbb{E}[\![\, e_1 \,]\!]\rho, v_2 \in \mathbb{E}[\![\, e_2 \,]\!]\rho \,\}$$
$$\mathbb{E}[\![\, -e \,]\!]\rho \stackrel{\text{def}}{=} \{\, -v \mid v \in \mathbb{E}[\![\, e \,]\!]\rho \,\}$$
$$\mathbb{E}[\![\, [a,b] \,]\!]\rho \stackrel{\text{def}}{=} \{\, x \in \mathbb{R} \mid a \leq x \leq b \,\}$$

$$\mathbb{B}[\![\, bexpr \,]\!] : \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$$

$$\mathbb{B}[\![\, e_1 \bowtie e_2 \,]\!]S \stackrel{\text{def}}{=} \{\, \rho \in S \mid \exists v_1 \in \mathbb{E}[\![\, e_1 \,]\!]\rho,\ v_2 \in \mathbb{E}[\![\, e_2 \,]\!]\rho,\ v_1 \bowtie v_2 \,\}$$
$$\mathbb{B}[\![\, b_1 \vee b_2 \,]\!]S \stackrel{\text{def}}{=} \mathbb{B}[\![\, b_1 \,]\!]\, S \cup \mathbb{B}[\![\, b_2 \,]\!]S$$
$$\mathbb{B}[\![\, b_1 \wedge b_2 \,]\!]S \stackrel{\text{def}}{=} \mathbb{B}[\![\, b_1 \,]\!]\, S \cap \mathbb{B}[\![\, b_2 \,]\!]S$$

$$\mathbb{S}[\![\, stat \,]\!] : \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$$

$$\mathbb{S}[\![\, V \leftarrow e \,]\!]S \stackrel{\text{def}}{=} \{\, \rho[V \mapsto v] \mid \rho \in S,\ v \in \mathbb{E}[\![\, e \,]\!]\rho \,\}$$
$$\mathbb{S}[\![\, \textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 \,]\!]S \stackrel{\text{def}}{=} \mathbb{S}[\![\, s_1 \,]\!]\, (\mathbb{B}[\![\, b \,]\!]S) \cup \mathbb{S}[\![\, s_2 \,]\!](\mathbb{B}[\![\, \neg b \,]\!]S)$$
$$\mathbb{S}[\![\, s_1; s_2 \,]\!]S \stackrel{\text{def}}{=} \mathbb{S}[\![\, s_2 \,]\!]\, (\mathbb{S}[\![\, s_1 \,]\!]S)$$

Fig. 5. Language semantics.

$$\mathbb{S}^\sharp[\![\, stat \,]\!] : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$$

$$\mathbb{S}^\sharp[\![\, V \leftarrow e \,]\!]S^\sharp \stackrel{\text{def}}{=} \mathfrak{assign}^\sharp(V, e, S^\sharp)$$
$$\mathbb{S}^\sharp[\![\, \textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 \,]\!]S^\sharp \stackrel{\text{def}}{=} \mathbb{S}^\sharp[\![\, s_1 \,]\!]\, (\mathfrak{guard}^\sharp(b, S^\sharp)) \cup^\sharp \mathbb{S}^\sharp[\![\, s_2 \,]\!](\mathfrak{guard}^\sharp(\neg b, S^\sharp))$$
$$\mathbb{S}^\sharp[\![\, s_1; s_2 \,]\!]S^\sharp \stackrel{\text{def}}{=} \mathbb{S}^\sharp[\![\, s_2 \,]\!]\, (\mathbb{S}^\sharp[\![\, s_1 \,]\!]S^\sharp)$$

$$\mathbb{B}^\sharp[\![\, bexpr \,]\!](S^\sharp) \stackrel{\text{def}}{=} \mathfrak{guard}^\sharp(bexpr, S^\sharp)$$

$$\mathbb{P}^\sharp(S^\sharp) \stackrel{\text{def}}{=} \mathbb{B}^\sharp[\![\, entry \,]\!]\, \top^\sharp \cup^\sharp \mathbb{S}^\sharp[\![\, body \,]\!]S^\sharp$$

Fig. 6. Abstract semantics.

We will abbreviate the problem statement as $\mathrm{lfp}\,\mathbb{P} \subseteq I$.

We can now formalize the notion of invariant and inductive invariant.

1. An *invariant* is any set I such that $\mathrm{lfp}\,\mathbb{P} \subseteq I$.
2. An *inductive invariant* is any set I such that $\mathbb{P}(I) \subseteq I$.

By Tarski's Theorem [29], $\mathbb{P}(I) \subseteq I$ implies $\mathrm{lfp}\,\mathbb{P} \subseteq I$. Moreover, $\mathrm{lfp}\,\mathbb{P}$ is the smallest invariant, and it is also an inductive invariant. Because computing or approximating $\mathbb{P}(I)$ is generally much easier than $\mathrm{lfp}\,\mathbb{P}$, it is much easier to check that a property is an inductive invariant than to check that it is an invariant.

3.2 Abstract Interpretation

The key idea underlying abstract interpretation is to replace each operation in $\mathcal{P}(\mathcal{E})$ with an operation that works in an abstract domain $\mathcal{D}^\sharp$ of properties. Each abstract element $S^\sharp \in \mathcal{D}^\sharp$ represents a set of points $\gamma(S^\sharp)$ through a concretization function $\gamma : \mathcal{D}^\sharp \to \mathcal{P}(\mathcal{E})$. Additionally, we assume that there exist abstract primitives $\mathfrak{assign}^\sharp(V, expr, S^\sharp)$, $\mathfrak{guard}^\sharp(bexpr, S^\sharp)$, $\cup^\sharp$, $\cap^\sharp$, $\bot^\sharp$, $\top^\sharp \in \mathcal{D}^\sharp$ that model, respectively, the effect of $\mathbb{S}[\![\, V \leftarrow expr \,]\!]$, $\mathbb{B}[\![\, bexpr \,]\!]$, $\cup$, $\cap$, $\emptyset$, and $\mathcal{E}$. Then, the semantics of Fig. 5 can be abstracted as shown in Fig. 6.

Once an inductive invariant is found by abstract interpretation, it is a simple matter to check whether the abstract inductive invariant entails the candidate

```
solutions ← ∅
toExplore ← ∅
push S into toExplore
while toExplore≠ ∅ do
    b ← pop(toExplore)
    b ← Hull-Consistency(b)
    if b = ∅ then nothing
    else if b contains only solutions then solutions ← solutions ∪ { b }
    else if b is small enough then solutions ← solutions ∪ { b }
    else
        split b in half along the largest dimension into b1 and b2
        push b1 and b2 into toExplore
done
```

Fig. 7. Continuous solver, from [24].

invariant I. Note that abstract interpretation does not require any knowledge of the invariant of interest to infer an inductive invariant, which is not a property enjoyed by our algorithm (or, at least, its first incarnation in Sec. 4; Sec. 5.2 will propose solutions to alleviate this limitation).

3.3 Continuous Constraint Solving

A constraint-satisfaction problem is defined by (i) a set of variables $V_1, \ldots, V_n$; (ii) a search space S given by a domain $D_1, \ldots, D_n$ for each variable; and (iii) a set of constraints $\phi_1, \ldots, \phi_p$. Because we are interested in a real-valued program semantics, we focus on continuous constraints: each constraint ϕ_i is a Boolean expression in our language (*bexpr* in Fig. 4). Moreover, each domain D_i is an interval of reals, so that the search space S is a box. The problem to be solved is to enumerate all variable valuations in the search space that satisfy every ϕ_i, i.e., to compute $\mathbb{B}[\![\wedge_i \phi_i]\!]S$. Because the solution set cannot generally be enumerated exactly, continuous solvers compute a collection of boxes with floating-point bounds that contain all solutions and tightly fit the solution set (i.e., contains as few non-solutions as possible). Such a solver alternates two kinds of steps:

1. *Propagation steps.* These exploit constraints to reduce the domains of variables by removing values that cannot participate in a solution. Ultimately, the goal is to achieve *consistency*, when no more values can be removed. Several notions of consistency exist. We use here so-called *hull consistency*, where domains are intervals and they are consistent when their bounds cannot be tightened anymore without possibly losing a solution.
2. *Splitting steps.* When domains cannot be reduced anymore, the solver performs an assumption: it splits a domain and continues searching in that reduced search space. The search proceeds, alternating propagation and splits, until the search space contains no solution, only solutions, or shrinks below a user-specified size. Backtracking is then used to explore other assumptions.

This algorithm is sketched in Fig. 7. It maintains a collection of search spaces yet to explore and a set of solution boxes. The algorithm always terminates, at which point every box either contains only solutions or meets the user-specified size limit, and every solution is accounted for (i.e., belongs to a box).

There are strong connections with abstract interpretation. We observed in [24] that the classic propagator for hull consistency for a constraint ϕ_i, so-called HC-4 [3], is very similar to the classic algorithm for $\mathfrak{guard}^\sharp(\phi_i, S)$ in the interval abstract domain. When there are several constraints, the propagators for each constraint are applied in turn, until a fixpoint is reached; this approach is similar to Granger's method of local iterations [15] used in abstract interpretation. We went further in [24] and showed that the algorithm of Fig. 7 is an instance of a general algorithm that is parameterized by an arbitrary numeric abstract domain. While standard continuous constraint programming corresponds to the interval domain, we established that there are benefits from using relational domains instead, such as octagons [20]. In abstract-interpretation terminology, this algorithm computes decreasing iterations in the disjunctive completion of the abstract domain. These iterations can be interpreted as an iterative over-approximation of a fixpoint: $\mathrm{gfp}\,\mathbb{C}$ where $\mathbb{C}(X) \stackrel{\text{def}}{=} \mathbb{B}[\![\wedge_i \phi_i]\!](X \cap S)$, and we can check easily that this concrete fixpoint $\mathrm{gfp}\,\mathbb{C}$ indeed equals $\mathbb{B}[\![\wedge_i \phi_i]\!]S$.

In the following, we will adapt this algorithm to over-approximate a program's semantic fixpoint, $\mathrm{lfp}\,\mathbb{P}$, instead (Eqn. (1)). There are two main differences between these fixpoints. First, constraint programming approximates a greatest fixpoint instead of a least fixpoint. Second, and less obvious, is that constraint programming is limited to approximating $\mathrm{gfp}\,\mathbb{C}$, where $\mathbb{C}$ is reductive, i.e., $\forall S : \mathbb{C}(S) \subseteq S$. It relies on the fact that, at every solving step, the join of all the boxes (whether solutions or yet to be explored) already covers all of the solutions. In a sense, it starts with a trivial covering of the solution, the entire search space, and refines it to provide a better (tighter) covering of the solution. In contrast, when we approximate $\mathrm{lfp}\,\mathbb{P}$, $\mathbb{P}$ is a join morphism, but is often neither extensive nor reductive. Moreover, the search will start with an invariant that may not be an inductive invariant.

4 Inductive-Invariant Inference

We now present our algorithm for inductive-invariant inference. We assume that we have the following abstract elements and computable abstract functions in the interval domain $\mathcal{D}^\sharp$ (cf. Sec. 3.2 and Fig. 6):

- $E^\sharp \stackrel{\text{def}}{=} \mathbb{B}^\sharp[\![\mathit{entry}]\!]\,\top^\sharp \in \mathcal{D}^\sharp$: the abstraction of the entry states
- $F^\sharp \stackrel{\text{def}}{=} \mathbb{S}^\sharp[\![\mathit{body}]\!] : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$: the abstraction of the loop body
- $I^\sharp \in \mathcal{D}^\sharp$: an abstract *under*-approximation of the invariant I we seek, i.e., $\gamma(I^\sharp) \subseteq I$ (in general the equality holds). We assume that $\gamma(E^\sharp) \subseteq \gamma(I^\sharp)$; otherwise, no subset of $\gamma(I^\sharp)$ can contain $\gamma(E^\sharp)$ and be an inductive invariant.

We will manipulate a set of boxes, $\mathfrak{S}^\sharp \in \mathcal{P}(\mathcal{D}^\sharp)$. Similar to disjunctive completion, we define the concretization $\gamma(\mathfrak{S}^\sharp)$ of a set of boxes $\mathfrak{S}^\sharp$ as the (concrete) union of their individual concretizations:

$$\gamma(\mathfrak{S}^\sharp) \stackrel{\text{def}}{=} \cup \{\, \gamma(S^\sharp) \mid S^\sharp \in \mathfrak{S}^\sharp \,\}. \tag{2}$$

We use $\gamma F^\sharp(\mathfrak{S}^\sharp)$ to denote the (concrete) union of the box-wise application of the abstract operator $F^\sharp$ on $\mathfrak{S}^\sharp$:

$$\gamma F^\sharp(\mathfrak{S}^\sharp) \stackrel{\text{def}}{=} \cup \{\, \gamma(F^\sharp(S^\sharp)) \mid S^\sharp \in \mathfrak{S}^\sharp \,\}. \tag{3}$$

Our goal is to find a set of boxes $\mathfrak{S}^\sharp$ that satisfies the following properties:

Property 1.
1. $\gamma(E^\sharp) \subseteq \gamma(\mathfrak{S}^\sharp)$ *($\mathfrak{S}^\sharp$ contains the entry)*
2. $\gamma(\mathfrak{S}^\sharp) \subseteq \gamma(I^\sharp)$ *($\mathfrak{S}^\sharp$ entails the invariant)*
3. $\gamma F^\sharp(\mathfrak{S}^\sharp) \subseteq \gamma(\mathfrak{S}^\sharp)$ *($\mathfrak{S}^\sharp$ is inductive)*

Prop. 1 is indeed sufficient to ensure we have found our inductive invariant.

Theorem 1.
$\gamma(\mathfrak{S}^\sharp)$ satisfying Props. 1.1–1.3 is an inductive invariant that implies I.

Proof. See the appendix of the technical report version of this paper [21]. □

In addition to Prop. 1, we will ensure that the boxes in $\mathfrak{S}^\sharp$ do not overlap. Because we use the interval abstract domain, which allows non-strict inequality constraints only, we do not enforce that $\mathfrak{S}^\sharp$ forms a partition. That is, we may have $\gamma(S_1^\sharp) \cap \gamma(S_2^\sharp) \neq \emptyset$ for $S_1^\sharp \neq S_2^\sharp \in \mathfrak{S}^\sharp$; however, intersecting boxes have an intersection of null volume, $\mathrm{vol}(\gamma(S_1^\sharp) \cap \gamma(S_2^\sharp)) = 0$, where $\mathrm{vol}(S^\sharp)$ is the volume of a box $S^\sharp$.

In a manner similar to a constraint-solving algorithm, our algorithm starts with a single box $\mathfrak{S}^\sharp \stackrel{\text{def}}{=} \{I^\sharp\}$, here containing the invariant to prove, and iteratively selects a box to shrink, split, or discard, until Props. 1.1–1.3 are satisfied. Prop. 1.1 (entry containment) holds when the algorithm starts, and we take care never to remove box parts that intersect $E^\sharp$ from $\mathfrak{S}^\sharp$. Prop. 1.2 (invariant entailment) also holds when the algorithm starts, and naturally holds at every step of the algorithm because we never add any point to $\gamma(\mathfrak{S}^\sharp)$. Prop. 1.3 (inductiveness) does *not* hold when the algorithm begins, and it is the main property to establish. We will start by presenting a few useful operations on individual boxes, before presenting our algorithm in Sec. 4.2.

4.1 Box operations

Box classification. To select a box to handle in $\mathfrak{S}^\sharp$, we must first classify the boxes. We say that $S^\sharp \in \mathfrak{S}^\sharp$ is:
- *necessary* if $\gamma(S^\sharp) \cap \gamma(E^\sharp) \neq \emptyset$;
- *benign* if $\gamma(F^\sharp(S^\sharp)) \subseteq \gamma(\mathfrak{S}^\sharp)$;
- *doomed*[4] if $\gamma(F^\sharp(S^\sharp)) \cap \gamma(\mathfrak{S}^\sharp) = \emptyset$;
- *useful* if $\gamma(S^\sharp) \cap \gamma F^\sharp(\mathfrak{S}^\sharp) \neq \emptyset$.

A necessary box contains some point from the entry $E^\sharp$, and thus cannot be completely discarded. A benign box has its image under $F^\sharp$ completely covered by $\gamma(\mathfrak{S}^\sharp)$, and so does not impede inductiveness. Our goal is to make all boxes benign. In contrast, a doomed box prevents inductiveness, and no shrinking or splitting operation on this or another box will make it benign. A useful box $S^\sharp$ intersects the image of a box $S_0^\sharp$ in $\mathfrak{S}^\sharp$, i.e., $S^\sharp$ helps make $S_0^\sharp$ benign and therefore $S^\sharp$ is worth keeping.

[4] If $F^\sharp(S^\sharp)$ is empty, then $S^\sharp$ satisfies our definition of benign and our definition of doomed; in this case we consider $S^\sharp$ benign, not doomed.

Coverage. In addition to these qualitative criteria, we define a quantitative measure of coverage:

$$\text{coverage}(S^\sharp, \mathfrak{S}^\sharp) \stackrel{\text{def}}{=} \frac{\sum \{ \text{vol}(\gamma(F^\sharp(S^\sharp)) \cap \gamma(T^\sharp)) \mid T^\sharp \in \mathfrak{S}^\sharp \}}{\text{vol}(\gamma(F^\sharp(S^\sharp)))}. \tag{4}$$

The coverage is a measure, in $[0, 1]$, of how benign a box is. A value of 1 indicates that the box is benign, while a value of 0 indicates that the box is doomed. Because our goal is to make all boxes benign, the algorithm will consider first the boxes with the least coverage, i.e., which require the most work to become benign. If a box is doomed, there is no hope for it to become benign, and it should be discarded. Similarly, if a box's coverage is too small, there is little hope it might contain a benign sub-box, and—as a heuristic—the algorithm could consider discarding it. For this reason, the algorithm is parameterized by a coverage cut-off value ϵ_c.

Tightening. Given a box $S^\sharp \in \mathfrak{S}^\sharp$, not all parts of $\gamma(S^\sharp)$ are equally useful. Using reasoning similar to the notion of consistency in constraint solvers, we can tighten the box. In our case, we remove the parts that do not make it necessary (intersecting $E^\sharp$) nor useful (intersecting $\gamma F^\sharp(\mathfrak{S}^\sharp)$). The box $S^\sharp$ is replaced with:

$$\text{tighten}(S^\sharp, \mathfrak{S}^\sharp) \stackrel{\text{def}}{=} (S^\sharp \cap^\sharp E^\sharp) \cup^\sharp (\cup^\sharp \{ S^\sharp \cap^\sharp F^\sharp(T^\sharp) \mid T^\sharp \in \mathfrak{S}^\sharp \}), \tag{5}$$

which first gathers all the useful and necessary parts from $S^\sharp$, and then joins them in the abstract domain to obtain a box that contains all those parts. Because the $\cup^\sharp$ operator of the interval domain is optimal, $\text{tighten}(S^\sharp)$ is the smallest box containing those parts. Replacing $S^\sharp$ with $\text{tighten}(S^\sharp, \mathfrak{S}^\sharp)$ in $\mathfrak{S}^\sharp$ has precision benefits. Because we keep useful parts, $\forall T^\sharp \neq S^\sharp : \text{coverage}(T^\sharp, \mathfrak{S}^\sharp)$ remains unchanged while $F^\sharp(S^\sharp)$ decreases—assuming that when $S^\sharp$ shrinks so does $F^\sharp(S^\sharp)$.[5]

Splitting. As in constraint solvers, we define the *size* $\text{size}(S^\sharp)$ of a box $S^\sharp \stackrel{\text{def}}{=} [a_1, b_1] \times \cdots \times [a_n, b_n]$ as the maximum width among all variables, i.e.:

$$\text{size}(S^\sharp) = \max \{ b_i - a_i \mid i \in [1, n] \}. \tag{6}$$

Box splitting consists of replacing a box $S^\sharp \stackrel{\text{def}}{=} [a_1, b_1] \times \cdots \times [a_n, b_n] \in \mathfrak{S}^\sharp$ with two boxes: $L^\sharp \stackrel{\text{def}}{=} [a_1, b_1] \times \cdots \times [a_i, c] \times \cdots \times [a_n, b_n]$ and $U^\sharp \stackrel{\text{def}}{=} [a_1, b_1] \times \cdots \times [c, b_i] \times \cdots \times [a_n, b_n]$, where $c = (a_i + b_i)/2$ is the middle of the interval $[a_i, b_i]$, and i is such that $b_i - a_i = \text{size}(S^\sharp)$, i.e., it is an interval whose size equals the size of the box. That way, a sequence of splits will reduce a box's size. We will refrain from splitting boxes below a certain size-parameter cut-off ϵ_s, and we are guaranteed to reach a size less than ϵ_s after finitely many splits.

Splitting a box also carries precision benefits. In general, abstract functions $F^\sharp$ are sub-join morphisms: $\gamma(F^\sharp(L^\sharp)) \cup \gamma(F^\sharp(U^\sharp)) \subseteq \gamma(F^\sharp(L^\sharp \cup^\sharp U^\sharp)) = \gamma(F^\sharp(S^\sharp))$. Splitting $S^\sharp$ corresponds to a case analysis, where the effect of the

[5] See the note about monotonicity on p. 16.

loop body is analyzed separately for $V_i \leq c$ and $V_i \geq c$, which is at the core of partitioning techniques used in abstract interpretation. While replacing $S^\sharp$ with $\{L^\sharp, U^\sharp\}$ in $\mathfrak{S}^\sharp$ does not change $\gamma(\mathfrak{S}^\sharp)$, it is likely to shrink $\gamma F^\sharp(\mathfrak{S}^\sharp)$, helping Prop. 1.3 to hold. Further improvements can be achieved by tightening $L^\sharp$ and $U^\sharp$ after the split.

Discarding. Constraint solvers only discard boxes when they do not contain any solution. Likewise, we can discard boxes that are neither necessary (no intersection with $E^\sharp$) nor useful (no intersection with $\gamma F^\sharp(\mathfrak{S}^\sharp)$). However, we can go further, and also remove useful boxes as well. On the one hand, this may shrink $\gamma F^\sharp(\mathfrak{S}^\sharp)$, and help Prop. 1.3 to hold. On the other hand, given $T^\sharp$ for which $S^\sharp$ is useful, i.e., $\gamma(F^\sharp(T^\sharp)) \cap \gamma(S^\sharp) \neq \emptyset$, it will strictly reduce $T^\sharp$'s coverage, and possibly make $T^\sharp$ not benign anymore. Nevertheless, in this latter case, it is often possible to recover from this situation later, by splitting, shrinking, or discarding $T^\sharp$ itself. The possibility to remove boxes that are unlikely to be in any inductive invariant is a particularly interesting heuristic to apply after the algorithm reaches the cut-off threshold for size or coverage. Such a removal will trigger the removal of parts of boxes whose image intersects the removed box, until—one hopes—after a cascade of similar effects, the algorithm identifies a core of boxes that forms an inductive invariant.

4.2 Algorithm

Our algorithm is presented in Fig. 8. It maintains, in $\mathfrak{S}^\sharp$, a set of boxes to explore, initialized with the candidate invariant $I^\sharp$ of interest. Then, while there are boxes to explore, the box $S^\sharp$ with smallest coverage (Eqn. (4)) is removed from $\mathfrak{S}^\sharp$. If $S^\sharp$ has a coverage of 1, then so have the remaining boxes in $\mathfrak{S}^\sharp$; hence all of the boxes are benign and the algorithm has found an inductive invariant: we add back $S^\sharp$ to $\mathfrak{S}^\sharp$ and return that set of boxes. Otherwise, some work is performed to help make $S^\sharp$ benign. In the general case, where $S^\sharp$ is not necessary (does not intersect $E^\sharp$), we are free to discard $S^\sharp$, which we do if it is useless, too small to be split, or its coverage makes it unlikely to become benign; otherwise, we split $S^\sharp$. When $S^\sharp$ is necessary, however, we refrain from removing it. Hence we split it, unless it is too small and we cannot split it anymore. At this point there is no way to satisfy all of Prop. 1.1, Prop. 1.3 and the user's specified minimum box size; consequently, the algorithm fails.

Normally, the **search** function always returns through either the success or failure **return** statements in the loop body. Normal loop exit corresponds to the corner case where all boxes have been discarded, which means that no box was ever necessary, i.e., $\mathbb{B}[\![\, \mathit{entry} \,]\!]\mathcal{E} = \emptyset$; we return $\emptyset$ in that case, because it is indeed an inductive invariant for loops with an empty entry state set.

The splitting procedure takes care to tighten the two sub-boxes $S_1^\sharp$ and $S_2^\sharp$ generated from the split. **split** is always called with a box of non-zero size, and it generates boxes smaller than $S^\sharp$.

Note that, given a box $S^\sharp$, it is possible that $\gamma(F^\sharp(S^\sharp)) \cap \gamma(S^\sharp) \neq \emptyset$, i.e., a box is useful to itself. For this reason, after removing $S^\sharp$ from $\mathfrak{S}^\sharp$, we take care

```
search(E♯, F♯, I♯):
    𝔖♯ ← {I♯}
    while 𝔖♯ ≠ ∅ do
        S♯ ← popMinCoverage(𝔖♯)
        if coverage(S♯, 𝔖♯ ∪ {S♯}) = 1 then return 𝔖♯ ∪ {S♯}
        else if S♯ is not necessary then
            if S♯ is not useful or size(S♯) < ε_s or coverage(S♯, 𝔖♯ ∪ {S♯}) < ε_c
            then discard(S♯)
            else split(S♯)
        else
            if size(S♯) < ε_s then return failed
            else split(S♯)
    done
    return ∅

split(S♯):
    (S₁♯, S₂♯) ← S♯ split in half along the largest dimension
    S₁♯ ← tighten(S₁♯, 𝔖♯ ∪ {S₁♯, S₂♯})
    S₂♯ ← tighten(S₂♯, 𝔖♯ ∪ {S₁♯, S₂♯})
    push S₁♯ and S₂♯ into 𝔖♯
```

Fig. 8. Inductive invariant search algorithm.

to compute the coverage as coverage$(S^\sharp, \mathfrak{S}^\sharp \cup \{S^\sharp\})$ and not as coverage$(S^\sharp, \mathfrak{S}^\sharp)$, and similarly after a split.

The following theorem states the correctness of the algorithm:

Theorem 2. search *always terminates, either with a failure, or returns a set* $\mathfrak{S}^\sharp$ *of boxes satisfying Props. 1.1–1.3.*

Proof. See the appendix of the technical report version of this paper [21]. □

Failure. It is important to note that, unlike constraint solvers, but similarly to iteration with widening, the algorithm may fail to find an inductive invariant, even if there exists one that can be represented with boxes whose size is greater than ϵ_s. In our case, the primary cause of failure is a bad decision to discard a useful box that actually intersects the smallest inductive invariant, a fact we cannot foresee when we decide to discard it. In Sec. 5.3, we present a modified version of the algorithm that handles this problem by changing the rules for discarding and tightening boxes, in exchange for a performance cost. Another method to avoid discarding such boxes is to lower the values of ϵ_s and ϵ_c, but this change can also make the algorithm slower, because it can now spend more time splitting boxes that cannot be part of an inductive invariant. Thus, another idea for future work is to provide more clever rules for discarding boxes, and using adaptive ϵ cut-off values. We discuss one such a technique in Sec. 5.2.

Example. Fig. 9 presents the evolution of $\mathfrak{S}^\sharp$ for the program from Fig. 2 through the execution of the algorithm. The benign boxes are shown in blue (dark gray), while the non-benign ones are shown in red (lighter gray). The

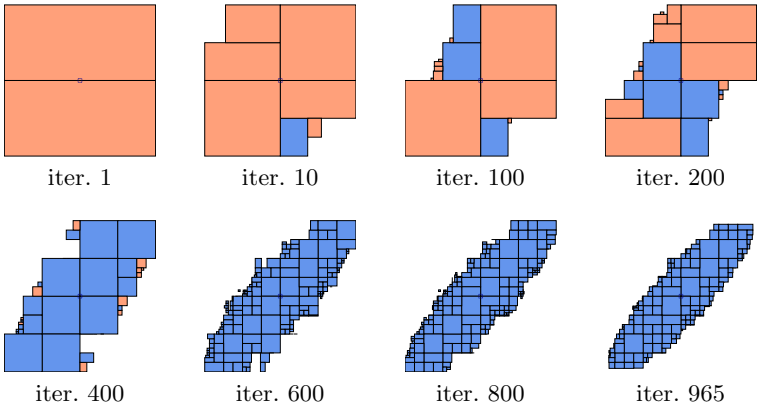


Fig. 9. Various stages of the analysis of the program from Fig. 2.

small dotted box at the center of each figure is the entry state $E^\sharp = [-0.1, 0.1] \times [-0.1, 0.1]$. The algorithm runs for 965 iterations in 76 ms before finding an inductive invariant included in $I^\sharp = [-4, 4] \times [-4, 4]$ and composed of 181 boxes, shown in Fig. 2.b. We use as cut-off values: $\epsilon_s = 0.01 \times \mathrm{size}(I^\sharp)$ and $\epsilon_c = 0.1 \times \mathrm{size}(I^\sharp)$. The size of the invariant box is $\mathrm{size}(I^\sharp) = 8$ as $I^\sharp \stackrel{\text{def}}{=} [-4, 4] \times [-4, 4]$.

4.3 Implementation details

The algorithm of Fig. 8 requires maintaining some information about boxes in $\mathfrak{S}^\sharp$, such as their coverage and whether they are benign or useful. It is important to note that modifying a single box in $\mathfrak{S}^\sharp$ may not only modify the coverage or class of the modified box, but also of other boxes in $\mathfrak{S}^\sharp$. We discuss here data structures to perform such updates efficiently, without scanning $\mathfrak{S}^\sharp$ entirely after each operation.

Partitioning. We enrich the algorithm state with a set $\mathfrak{P}^\sharp \in \mathcal{P}(\mathcal{D}^\sharp)$ of boxes that, similarly to $\mathfrak{S}^\sharp$, do not overlap, but—unlike $\mathfrak{S}^\sharp$—always covers the whole invariant space $I^\sharp$. Moreover, we ensure that every box in $\mathfrak{P}^\sharp$ contains at most one box from $\mathfrak{S}^\sharp$, hence, we maintain a *contents-of* function $\mathrm{cnt} : \mathfrak{P}^\sharp \to (\mathfrak{S}^\sharp \cup \{\emptyset\})$ indicating which box, if any, of $\mathfrak{S}^\sharp$ is contained in each part of $\mathfrak{P}^\sharp$. Because the algorithm can discard boxes from $\mathfrak{S}^\sharp$, some parts in $P^\sharp \in \mathfrak{P}^\sharp$ may have no box at all, in which case $\mathrm{cnt}(P^\sharp) = \emptyset$; otherwise $\gamma(\mathrm{cnt}(P^\sharp)) \supseteq \gamma(P^\sharp)$. This property can be easily ensured by splitting boxes in $\mathfrak{P}^\sharp$ whenever a box in $\mathfrak{S}^\sharp$ is split. When a box from $\mathfrak{S}^\sharp$ is tightened or discarded, no change in $\mathfrak{P}^\sharp$ is required.

We then maintain a map $\mathrm{post} : \mathfrak{S}^\sharp \to \mathcal{P}(\mathfrak{P}^\sharp)$ to indicate which parts of $\mathfrak{P}^\sharp$ intersect the image of a box $S^\sharp \in \mathfrak{S}^\sharp$:

$$\mathrm{post}(S^\sharp) \stackrel{\text{def}}{=} \{\, P^\sharp \in \mathfrak{P}^\sharp \mid \gamma(F^\sharp(S^\sharp)) \cap \gamma(P^\sharp) \neq \emptyset \,\}, \tag{7}$$

which is sufficient to compute the coverage and determine if a box is benign:

$$\mathrm{coverage}(S^\sharp) \stackrel{\text{def}}{=} \frac{\sum \{\, \mathrm{vol}(\gamma(F^\sharp(S^\sharp)) \cap \gamma(\mathrm{cnt}(P^\sharp))) \mid P^\sharp \in \mathrm{post}(S^\sharp) \,\}}{\mathrm{vol}(\gamma(F^\sharp(S^\sharp)))} \tag{8}$$

$$S^\sharp \text{ is benign} \iff \gamma(F^\sharp(S^\sharp)) \subseteq \gamma(I^\sharp) \wedge \forall P^\sharp \in \mathrm{post}(S^\sharp) : \gamma(P^\sharp) \cap \gamma(F^\sharp(S^\sharp)) \subseteq \gamma(\mathrm{cnt}(P^\sharp)) \ . \tag{9}$$

A box $S^\sharp$ is benign if, whenever $F^\sharp(S^\sharp)$ intersects some partition $P^\sharp \in \mathfrak{P}^\sharp$, their intersection is included in $\mathrm{cnt}(P^\sharp) \in \mathfrak{S}^\sharp$. Note, however, that this test only takes into account the part of $F^\sharp(S^\sharp)$ that is included in $\gamma(\mathfrak{P}^\sharp)$, i.e., in $\gamma(I^\sharp)$; hence, we additionally also check that $F^\sharp(S^\sharp)$ is included within $I^\sharp$.

Likewise, to compute tighten (Eqn. (5)) and determine whether a box is useful, it is sufficient to know, for each box $S^\sharp \in \mathfrak{S}^\sharp$, the boxes $T^\sharp \in \mathfrak{S}^\sharp$ whose image $F^\sharp(T^\sharp)$ intersects $S^\sharp$, which we denote by:

$$\mathrm{pre}(S^\sharp) \stackrel{\text{def}}{=} \{\, T^\sharp \in \mathfrak{S}^\sharp \mid \gamma(S^\sharp) \cap \gamma(F^\sharp(T^\sharp)) \neq \emptyset \,\} \ . \tag{10}$$

The sets $\mathrm{post}(S^\sharp)$ and $\mathrm{pre}(S^\sharp)$ are significantly smaller than $\mathfrak{S}^\sharp$, leading to efficient ways of recomputing coverage information, usefulness, etc. They are also quite cheap to maintain through box tightening, splitting, and discarding.

We rely on the fact that, apart from the initial box $I^\sharp$, every new box considered by the algorithm comes from a parent box by splitting or tightening, and is thus smaller than a box already in $\mathfrak{S}^\sharp$. Assuming for now that, when $S^\sharp$ is shrunk into $T^\sharp$, $F^\sharp(T^\sharp)$ is also smaller than $F^\sharp(S^\sharp)$ (see the note on monotonicity below), we see that $\mathrm{post}(T^\sharp)$ and $\mathrm{pre}(T^\sharp)$ are subsets, respectively, of $\mathrm{post}(S^\sharp)$ and $\mathrm{pre}(S^\sharp)$, so that it is sufficient to iterate over these sets and filter out elements that no longer belong to them. Finally, to quickly decide for which $S^\sharp$ we must update $\mathrm{post}(S^\sharp)$ and $\mathrm{pre}(S^\sharp)$ whenever some element in $\mathfrak{P}^\sharp$ or $\mathfrak{S}^\sharp$ is modified or discarded, we maintain post^{-1} and pre^{-1} as well.

Float implementation. While it is possible to use exact numbers—such as rationals with arbitrary precision—to represent interval bounds, more efficiency can be achieved using floating-point numbers. Floats suffer, however, from rounding error. It is straightforward to perform sound abstract interpretation with floating-point arithmetic. In particular, to construct a suitable $F^\sharp$ function from the program syntax, it is sufficient to round—in every computation—upper bounds towards $+\infty$ and lower bounds towards $-\infty$. In addition to the requirement that we have a sound $F^\sharp$, our algorithm requires that we can (i) compute the coverage of a box, and (ii) determine whether it is benign, necessary, or useful. Because intersection, inclusion, and emptiness checking are exact operations on float boxes, we can exactly determine whether a box is benign, necessary, or useful; it is also possible to compute a box size exactly.

In contrast, it is difficult to compute exactly the volume of a box using floating-point arithmetic, which is a necessary operation to compute coverage (Eqns. (4),(8)). Fortunately, our algorithm does not require exact coverage information to be correct; thus, we settle for an approximate value computed naively using Eqn. (8) with rounding errors. However, it is critical that we use the exact formula Eqn. (9), which uses exact inclusion checking, to determine whether a box is benign; this is almost equivalent to checking whether the coverage of a box equals 1, except that the coverage calculation is subject to floating-point

round-off error, so it is not safe to use here. This approach also requires changing, in Fig. 8, the test "**if** coverage$(b, \mathfrak{S}^\sharp \cup \{b\}) = 1$" into a test that all the boxes in $\mathfrak{S}^\sharp$ are benign, using Eqn. (9).

Note on monotonicity. Throughout this section, we assumed that replacing a box $S^\sharp$ with a smaller one $T^\sharp \subseteq^\sharp S^\sharp$ also results in a smaller image, i.e., $F^\sharp(T^\sharp) \subseteq^\sharp F^\sharp(S^\sharp)$. This property is obviously the case if $F^\sharp$ is monotonic, but the property does not hold in the general case (e.g., if $F^\sharp$ employs widening or reduced products). We now argue that we can handle the general case, by forcing images of boxes to decrease when the boxes decrease as follows: we remember with each box $S^\sharp$ its image $\mathrm{img}(S^\sharp)$; when $S^\sharp$ is replaced with a smaller box $T^\sharp$, we set its image to:

$$\mathrm{img}(T^\sharp) := F^\sharp(T^\sharp) \cap^\sharp \mathrm{img}(S^\sharp) \tag{11}$$

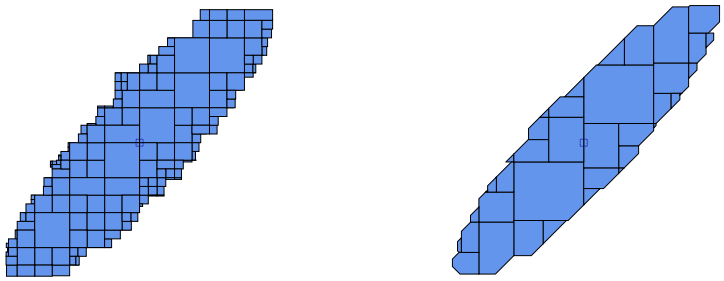
instead of $F^\sharp(T^\sharp)$. We rely again on the fact that each new box comes from a larger parent box, except the initial box $I^\sharp$, for which we set $\mathrm{img}(I^\sharp) \stackrel{\text{def}}{=} F^\sharp(I^\sharp)$. We note that because the concrete semantics $\mathbb{S}[\![\, body \,]\!]$ is monotonic, $\gamma(\mathrm{img}(T^\sharp)) \subseteq \mathbb{S}[\![\, body \,]\!](\gamma(T^\sharp))$; that is, the use of equation (11) provides as sound an abstraction of $\mathbb{S}[\![\, body \,]\!]$ as $F^\sharp$ but, in addition, it is monotonic. Hence, by replacing $F^\sharp(S^\sharp)$ with $\mathrm{img}(S^\sharp)$ in the definitions from Sec. 4.1, our algorithm is correct, even if $F^\sharp$ is not monotonic.

5 Extensions

5.1 Octagonal invariants

Following the custom from constraint solvers for continuous constraints, the algorithm presented in Sec. 4 is based on boxes: the loop body $F^\sharp$ is evaluated in the interval abstract domain. However, we showed in [24] that, in constraint solvers, boxes can be replaced with arbitrary numeric abstract domains, for instance relational domains such as polyhedra or octagons, provided that certain needed operators are provided. Similarly, the algorithm from the present paper can be generalized to use an arbitrary abstract domain $\mathcal{D}^\sharp$. Existing domains readily provide abstractions $F^\sharp$ of the loop body, together with the abstractions of $\cup$, $\cap$, and $\subseteq$ required for tightening as well as testing whether an abstract element is necessary, benign, or useful. We only need three extra operations not found in classical abstract domains for our algorithm: the volume vol (used to compute the coverage), the size operation, and a split operation. As discussed in Sec. 4.3, it is not necessary for vol to compute the exact volume, as long as it is possible to determine exactly whether an abstract element is benign, which is the case using Eqn. (9). Likewise, we have a great deal of freedom in defining the size and split operations. The only requirement is that, to ensure the termination of the algorithm, after a sufficient number of splits are performed, an abstract element will be reduced below an arbitrary size.

Example. Consider the octagon domain [20], which is more expressive than intervals because it can represent relational information of the form $\pm X \pm Y \leq c$, with two variables and unit coefficients. We choose to approximate the volume of an octagon as the volume of its bounding box, which is far easier to compute



(a) 181 boxes, 965 iterations, 76 ms (b) 42 octagons, 224 iterations, 89 ms

Fig. 10. Abstract-domain choice: box solving (a) versus octagon solving (b).

than the actual volume of the octagon. Likewise, to create a simple and efficient implementation, we use the same split operation as for boxes, splitting only along an axis, and use the "size" of the bounding box as the "size" of an octagon. As a consequence, the elements of the partition $\mathfrak{P}^\sharp$ remain boxes. Nevertheless, the elements of $\mathfrak{S}^\sharp$ can be more complex octagons, due to tightening. We refer the reader to [24] for more advanced size and split operators for octagons.

Figure 10 compares the result of our algorithm on the program of Fig. 2 using intervals and using octagons. Inference with intervals takes 965 iterations and 76 ms and produces an inductive invariant composed of 181 parts, while, using octagons, it takes 224 iterations and 89 ms, and the inductive invariant is composed of only 42 parts. As can be seen in Fig. 10, the slant of octagons allows a much tighter fit to an ellipsoid, with far fewer elements. Octagons are also slightly slower: they require far fewer iterations and elements, but the manipulation of a single octagon is more costly than that of a single box.

Domain choice. The choice of abstract domain $\mathcal{D}^\sharp$ may affect whether or not an inductive invariant is found, and it may also affect the cost of running the algorithm; this situation is somewhat similar to what happens in abstract interpretation. Depending on the problem, a more expressive domain, such as octagons, can result in a faster or slower analysis (see Sec. 6).

5.2 Invariant refinement

The algorithm assumes that a candidate invariant I is provided, and by removing points only looks for an inductive invariant sufficient to prove that I holds. It over-approximates a least fixpoint from above and, as a consequence, the inductive invariant it finds may not be the smallest one. In fact, because it stops as soon as an inductive invariant is found, it is more likely to output one of the greatest inductive invariants contained in I. In contrast, the iteration-with-widening technique used in abstract interpretation over-approximates the least fixpoint, but approaches it from below. Although such iterations may also fail to find the least fixpoint, even when decreasing iterations with narrowing are used,[6]

[6] This situation happens when the widening overshoots and settles above a fixpoint that is not the least one, because decreasing iterations can only refine up to the immediately smaller fixpoint, and not skip below a fixpoint.

they are more likely to find a small inductive invariant, because they start small and increase gradually.

We propose here several methods to improve the result of our algorithm towards a better inductive invariant. They also mitigate the need for the user to provide a candidate invariant I, which was a drawback of our method compared to the iteration-with-widening method approaching from below. In fact, we can start the algorithm with a large box I (such as the whole range of a data type) and rely on the methods given below to discover an inductive invariant whose bounding box is much smaller than I, effectively inferring an interval invariant, alongside a more complex inductive invariant implying that invariant.

Global tightening. The base algorithm of Fig. 8 only applies tightening (Eqn. (5)) to the two boxes created after each split. This approach is not sufficient to ensure that all the boxes in $\mathfrak{S}^\sharp$ are as tight as possible. Indeed, shrinking or removing a box $S^\sharp$ also causes $\gamma F^\sharp(\mathfrak{S}^\sharp)$ to shrink. Hence, boxes in $\mathfrak{S}^\sharp$ that contained parts of $\gamma F^\sharp(\mathfrak{S}^\sharp)$ that have been removed may be tightened some more, enabling further tightening in a cascading effect. Hence, to improve our result, we may iteratively perform tightening of all the boxes in $\mathfrak{S}^\sharp$ until a fixpoint is reached. Because this operation may be costly, it is better to apply it rarely, such as after finding a first inductive invariant.

Reachability checking. The concrete semantics has the form $\mathrm{lfp}\,\lambda X.\, E \cup G(X)$, where $E \stackrel{\text{def}}{=} \mathbb{B}[\![\, \mathit{entry} \,]\!]\, \mathcal{E}$ is the entry state and $G \stackrel{\text{def}}{=} \mathbb{S}[\![\, \mathit{body} \,]\!]$ is the loop body. This fixpoint can also be seen as the limit $\cup_{i \geq 0} \mathbb{S}[\![\, \mathit{body} \,]\!]^i\, E$, i.e., the points reachable from the entry state after a sequence of loop iterations. To improve our result, we can thus remove parts of $\gamma(\mathfrak{S}^\sharp)$ that are definitely not reachable from the entry state. This test can be performed in the abstract domain by computing the transitive closure of $\mathrm{cnt} \circ \mathrm{post}$ (Eqn. (7)), starting from all the necessary boxes (i.e., those intersecting $E^\sharp$). This operation may also be costly and is better performed after an inductive invariant is found.

Shell peeling. Like iteration with narrowing, the methods given above are only effective at improving our abstraction of a *given* fixpoint, but are unable to reach an abstraction of a *smaller* fixpoint. The shell-peeling method builds on the reachability principle to discard boxes more aggressively, based on their likelihood of belonging to the most-precise inductive invariant. Intuitively, as imprecision accumulates by repeated applications of $F^\sharp$ from necessary boxes, boxes reachable in many steps are likely to contain the most spurious points. For each box $S^\sharp \in \mathfrak{S}^\sharp$, we compute its depth, i.e., the minimal value i such that $S^\sharp \cap^\sharp (\mathrm{cnt} \circ \mathrm{post})^i(T^\sharp) \neq \bot^\sharp$ for some $T^\sharp \in \mathfrak{S}^\sharp$ that intersects $E^\sharp$. Given the maximal depth $\Delta \stackrel{\text{def}}{=} \max\{\, \mathrm{depth}(S^\sharp) \mid S^\sharp \in \mathfrak{S}^\sharp \,\}$, we merely remove all the boxes with depth greater than $\Delta - \delta$, for some user-specified δ, i.e., the boxes farthest away from the entry state in terms of applications of $F^\sharp$. Because the resulting $\mathfrak{S}^\sharp$ is not likely to be inductive, we again run the algorithm of Fig. 8 to try to recover an inductive invariant.

Resplitting. As explained in Sec. 4.1, splitting boxes in $\mathfrak{S}^\sharp$ may, by itself, improve the precision because $F^\sharp$ is generally a sub-join morphism, and so, even

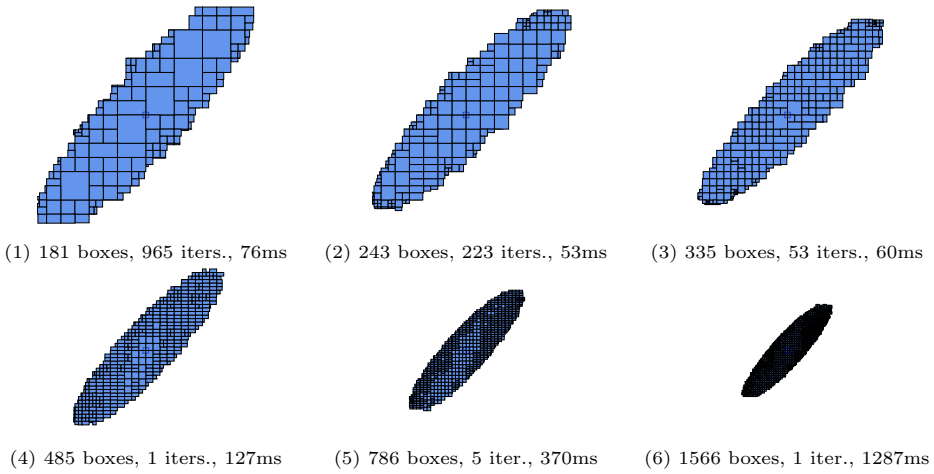
though it leaves $\gamma(\mathfrak{S}^\sharp)$ unchanged, $\gamma F^\sharp(\mathfrak{S}^\sharp)$ will decrease. As a consequence, some parts of $\mathfrak{S}^\sharp$ that were reachable from $E^\sharp$ through post may become unreachable. Hence, we suggest splitting boxes in $\mathfrak{S}^\sharp$ proactively, and prioritize the boxes $S^\sharp$ that are likely to reduce the size of $\text{post}(S^\sharp)$ (i.e., the number of boxes intersecting $F^\sharp(S^\sharp)$), before using other techniques such as global tightening or reachability checking. More precisely, we split all the boxes such that $|\text{post}(S^\sharp)|$ exceeds a user-defined threshold R.

Application. We apply all four methods on the example of Fig. 2 as follows. First, we apply the algorithm of Fig. 8, followed by global tightening and reachability checking. Then, we perform several rounds of the following steps: resplitting, shell peeling, applying the algorithm of Fig. 8, global tightening, followed by reachability checking. The result of this process is shown in Fig. 11. We set $\delta = 1$ for shell peeling, and $R = 12$ as the resplitting threshold. Moreover, each time the algorithm from Fig. 8 is run again, the ϵ values are halved, to account for the general reduction of box sizes due to resplitting and to allow finer-granularity box covering. Each refinement round results in a much tighter inductive invariant. We see a large gain in precision at the cost of higher computation time and a finer invariant decomposition (i.e., more and smaller boxes). Almost all the work goes into the four refinement methods, with just a few box-level iterations to re-establish inductiveness.

Failure recovery. The algorithm is parameterized with ϵ values, and, as discussed in Sec. 4, a too-high value may result in too many boxes being discarded too early, and a consequent failure to find an inductive invariant. For instance, when setting ϵ_s to $0.1 \times \text{size}(I^\sharp)$ instead of $0.01 \times \text{size}(I^\sharp)$, we fail to find an inductive invariant for the program from Fig. 2. A natural, but costly, solution would be to restart the algorithm from scratch with lower ϵ values. Alternatively, we can also apply the refinement techniques above. Given the failure output $\mathfrak{S}^\sharp$ of the algorithm from Fig. 8, which satisfies Prop. 1.1 and Prop. 1.2, but not Prop. 1.3, we iterate the following steps: global tightening, reachability checking, resplitting, and reapplying the algorithm of Fig. 8, always feeding the failure output of each round to the following one without resetting $\mathfrak{S}^\sharp$ to $\{I^\sharp\}$ until an inductive invariant is found. Using the same δ and R parameters as before, and also halving the ϵ values at each round, starting from $\epsilon_s = 0.1 \times \text{size}(I^\sharp)$, we find in 5 rounds an inductive invariant, and it is as good as the one in Fig. 11.5 which was found with $\epsilon_s = 0.01 \times \text{size}(I^\sharp)$.

5.3 Relative Completeness

Our algorithm searches for loop invariants in a particular abstract domain, namely, sets of boxes with size between ϵ_s and $\epsilon_s/2$. As noted in Sec. 4.2, however, the algorithm is not guaranteed to find an inductive invariant even if one exists in this abstract domain. In this section, we define a different abstract domain and present a modified version of the algorithm that has the following *relative-completeness* property [17]: if there is an inductive invariant in the abstract domain, and its inductiveness can be proven using boxes, then the modified algorithm will find an inductive invariant. Thus, when the modified algorithm



(1) 181 boxes, 965 iters., 76ms (2) 243 boxes, 223 iters., 53ms (3) 335 boxes, 53 iters., 60ms

(4) 485 boxes, 1 iters., 127ms (5) 786 boxes, 5 iter., 370ms (6) 1566 boxes, 1 iter., 1287ms

Fig. 11. Inductive-invariant refinement, using 6 rounds of tightening, reachability checking, shell peeling, resplitting, and the algorithm from Fig. 8. (Diagrams are to scale.)

fails to find an inductive invariant, it provides a stronger guarantee than the original algorithm, because a failure of the modified algorithm can only occur when there exists no inductive invariant in the abstract domain. The modified algorithm is slower than the original, however, so we have implemented it as a variant that can be used when completeness is more important than speed.

During a run of the algorithm, if no boxes are ever discarded or tightened, the algorithm's process of box-splitting, beginning with some (abstract) candidate invariant $I^\sharp$, will eventually produce a regular grid of boxes of minimal size—i.e., having size between ϵ_s and $\epsilon_s/2$. Denote by Q the set of all boxes that can be produced in such a run. In the two-dimensional case, the boxes of Q may be conceptualized as a binary space partition; it is almost like quadtree, except that boxes are actually split along one dimension at a time, so each node in the tree has two children, not four. The width of the minimal-size boxes of Q along dimension i is $max\{2^{-k}w_i \mid k \in \mathbb{N}, 2^{-k}w_i < \epsilon_s\}$, where w_i is the width of the given invariant $I^\sharp$ along dimension i. However, Q also contains the larger boxes (unions of the minimal-size ones) that were produced at intermediate stages of the splitting process. We define the abstract domain C_Q for the modified version of the algorithm to be the set of all non-overlapping sets of boxes from Q.

There are two reasons why the original algorithm, presented in Fig. 8, lacks the relative-completeness property. First, the algorithm discards a box $S^\sharp$ if its coverage is less than ϵ_c; however, it is possible that $S^\sharp$ intersects every inductive invariant, in which case discarding $S^\sharp$ guarantees that a failure will occur. Second, the tightening operation can replace a box $S^\sharp$ that is part of Q with a box $T^\sharp$ that is not part of Q. In many cases an inductive invariant can still be found; the invariant can contain boxes that are not part of Q, and the vertices of boxes in such an invariant might not correspond to the vertices of any box

from Q. However, it is also possible that $T^\sharp$, along with the other boxes of $\mathfrak{S}^\sharp$, cannot be refined into an inductive invariant.

For an example of the problem with tightening, see Fig. 12. Suppose that splitting $S^\sharp$ yields boxes $S_1^\sharp$ and $S_2^\sharp$, both having size below ϵ_s, such that $S_1^\sharp$ needs to be discarded, because it is not (and will never become) benign, and $S_2^\sharp$ needs to be kept, because it is necessary. Suppose that $T^\sharp$ is the result of applying tightening to $S^\sharp$ instead of splitting $S^\sharp$. It is possible that $T^\sharp$ has size below ϵ_s, and so cannot be split, even though it contains enough of both $S_1^\sharp$ and $S_2^\sharp$ so that it is necessary and yet it is also not benign (and will never become benign). Consequently, neither discarding nor keeping $T^\sharp$ will yield an inductive invariant. In this case, the choice to tighten $S^\sharp$ prevented the algorithm from finding an invariant that could have been found by splitting $S^\sharp$ instead.

Two simple modifications, which address the above two problems, allow the algorithm to have the relative-completeness property. First, we set $\epsilon_c = 0$, and second, we never perform tightening. For each of these two modifications, we have constructed an input program for which the modified algorithm finds an invariant, but the original algorithm does not. Note that there also exist programs for which the original algorithm can find an inductive invariant, but the modified algorithm cannot, because there is no inductive invariant in the modified abstract domain C_Q. As an optimization to the modified algorithm, which performs no tightening, we could reintroduce the tightening operation in a modified form: instead of tightening down to an arbitrary smaller box $T^\sharp$, we could tighten down to the smallest box in Q that contains $T^\sharp$. We present a proof of the relative-completeness property of the modified algorithm in [21].

6 Experiments

We have implemented a prototype analyzer, written in OCaml. It supports a simple toy language similar to Fig. 4, with only real-valued variables, no functions, and no nested loops. It implements our main algorithm (Fig. 8) with arbitrary numeric domains (Sec. 5.1). It also implements invariant-refinement and failure-recovery methods (Sec. 5.2). It is parameterized by a numeric abstract domain, and currently supports boxes with rational bounds, boxes with float bounds, as well as octagons with float bounds (through the Apron library [18]). It exploits the data structures described in Sec. 4.3 and is sound despite floating-point round-off errors. Floating-point arithmetic is used in the implementation.

The results of our experiments are shown in Fig. 13. We show, for the floating-point interval and octagon domains, the number of iterations and time until a

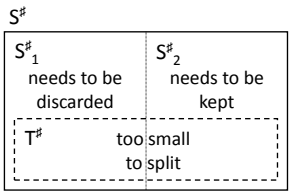


Fig. 12. If tightening is applied to a box $S^\sharp$, it may become too small to split in a case where splitting $S^\sharp$ is the only way to find an inductive invariant.

first inductive invariant is found, and the number of elements in each result.[7] We do not give details about the memory consumption for each example, because it always remains low (a few hundred megabytes). For most examples, we used an ϵ_s value of 0.05 or 0.01. We have analyzed a few simple loops, listed above the double line in Fig. 13:

– *Filter* is the second-order filter from Fig. 2. As discussed earlier, it performs well with intervals and octagons.

– *Linear* iterates $t = t + 1; \tau = \tau + [0.25, 0.5]$ from $t = \tau = 0$ until $t < 30$. To prove that $\tau < 30$, it is necessary to find a relation $\tau \leq 0.5 \times t$, which cannot be represented by intervals or octagons (without disjunctive completion). *Non-linear* is similar, but iterates $\tau = [1, 1.1] \times \tau$ starting from $\tau \in [0, 1]$, and also requires a non-linear inductive invariant: $\tau \leq 1.1^t$. These examples require very few iterations of our main algorithm, but rely heavily on resplitting, and hence output a large set of elements and have a relatively high run-time, on the order of a few seconds. When using octagons, the method times out after 300s.

– *Logistic map* iterates the logistic function $x_{n+1} = r \times x_n \times (1 - x_n)$, starting from $x \in [0.1, 0.9]$, for $r \in [1.5, 3.568]$. Our goal is to prove that x remains within $[0.1, 0.9]$ at all times. The choice of $r \in [1.5, 3.568]$ corresponds to various cases where the series oscillates between one or several non-zero values; it is beyond the zone where it converges towards 0 ($r \leq 1$), but before it becomes chaotic ($r \geq 3.57$). The inductive invariant covers the whole space $[0.1, 0.9] \times [1.5, 3.568]$ (hence, the number of elements equals the number of iterations), but needs to be partitioned so that the desired invariant can be proved in our linear abstract domains. Our algorithm performs this partitioning automatically.

– *Newton* iterates Newton's method on a polynomial taken from [11].

– The next two examples come from the conflict-driven learning method proposed in [11]: *Sine* and *Square root* compute mathematical functions through Taylor expansions. Unlike our previous examples, they do not perform an iterative computation, only a straight-line computation; nevertheless, they require partitioning of the input space to prove that the target invariant holds after the straight-line computation. These examples demonstrate that the algorithm is able to compute a strongest-postcondition of a straight-line computation, which helps to show how the algorithm could be generalized to handle programs with a mix of straight-line and looping code. In both cases, the inductive invariant matches closely the graph of the function (up to method error and rounding error), and hence provide a proof of functional correctness. As in the *Linear* and *Non-Linear* examples, these examples rely heavily on resplitting and tightening.

For more details about these examples, see the appendix of [21].

In a second batch of experiments, shown in Fig. 13 below the double line, we ran our algorithm on benchmark programs from the literature. These experiments were designed to answer the following questions:

1. Can our algorithm find invariants for programs studied in earlier work?
2. How fast is our algorithm when analyzing such programs?

[7] We include failure-recovery rounds from Sec. 5.2 when needed to find the first invariant, but do not perform any refinement rounds after an invariant is found.

Program	boxes			octagons		
	# elems.	# iters.	time (s)	# elems.	# iters.	time (s)
Filter	181	965	0.076	42	224	0.089
Linear	6734	10	2.08	–	–	–
Non-linear	5987	10	2.64	–	–	–
Logistic map	376	376	0.127	885	885	2.94
Newton	32	57	0.009	16	16	0.008
Sine	132	425	0.094	81	99	0.076
Square root	8	8	0.002	4	4	0.002
Lead-lag controller	44	46	0.327	30	30	65.6
Lead-lag controller (reset)	24	24	2.44	24	24	105.0
Lead-lag controller (saturate)	44	46	0.341	30	30	76.7
Harmonic oscillator	32	36	0.04	42	46	1.62
Harmonic oscillator (reset)	34	39	0.047	–	–	–
Harmonic oscillator (saturate)	4	4	0.035	3	3	0.44
Dampened oscillator	850	45400	1.49	929	45404	18.7
Dampened oscillator (reset)	925	57329	37.2	–	–	–
Dampened oscillator (saturate)	22	22	0.004	22	22	0.036
Filter2	9	160	0.01	5	5	0.003
Arrow-Hurwicz	80	1904	0.367	18	839	0.451

Fig. 13. Experimental results. For octagons, the timeout value was 300 seconds.

3. Are we able to verify some invariants found by other algorithms?

To answer these questions, we selected three programs, in three variations each, from [26], and two programs from [1]. For each of the programs from [26], the parameter $I^\sharp$ for our algorithm (i.e., the invariant that we attempted to verify) was the bounding box of an invariant described in either the text of the publication or its additional online materials. For the programs from [1], we chose small boxes for $I^\sharp$. We analyzed the Arrow-Hurwicz loop as a two-variable program by discarding the loop condition and the two variables u and v that are not needed in the body of the loop; the algorithm was able to verify that the variables x and y remain within the box $[-2, 2] \times [-2, 2]$. For Filter2, we showed that x and y remain within $[-0.2, 1] \times [-0.2, 1]$.

Our results show that we were able to find inductive invariants for all of these programs, often in less than half a second. In the case of the dampened oscillator, we were not able to verify the bounds described in the text of [26], but we were able to verify a different bounding box described in the additional online materials.

7 Related Work

A recent line of work has identified conceptual connections between SAT-solving and abstract interpretation [10,31], and has exploited these ideas to improve SAT-solving algorithms. That idea is similar in spirit to work described in [24], except that the latter focuses on other classes of algorithms—those used in continuous constraints programming—that work on geometric entities, such as boxes, instead of formulas.

As mentioned in Sec. 2, even when an inductive invariant can be represented in the chosen domain, approximation due to widening can prevent an abstract interpreter from finding any inductive invariant—let alone the best one. A certain amount of recent work has been devoted to improved algorithms for computing fixpoints in abstract domains. In particular, methods based on policy iteration

can guarantee, in certain cases, that the exact least fixpoint is computed [1]. The version of our algorithm presented in Sec. 5.3 uses a modified abstract domain and has the following *relative-completeness* property [17]: if there is an inductive invariant in the modified abstract domain, and its inductiveness can be proven using boxes, then the modified algorithm will find an inductive invariant.

Garg et al. [13] developed a framework for loop-invariant synthesis called ICE. ICE has a property that they call *robustness*, which is similar to relative completeness. They observed that many earlier algorithms consist of a teacher and a learner that communicate using a restricted protocol, and that this protocol sometimes prevents the algorithms from finding an invariant. In the restricted protocol, the teacher only communicates positive and negative examples of states, and the learner tries to find an invariant consistent with these. The authors propose an alternative protocol that also allows the teacher to communicate implication pairs (p, p'), to convey that state p can transition to state p'; this generalization allows the algorithm to postpone the decision of whether to include or exclude p and p' from the invariant, instead of arbitrarily—and perhaps incorrectly—choosing to classify p and p' as examples or counterexamples.

In our algorithm, the analogue of ICE's implication pairs, counterexamples, and examples is the set of boxes $\mathfrak{S}$ and their images. Where ICE would store a concrete implication pair to represent the fact that state p leads to state p', our algorithm would store a box containing state p along with the image of that box. As with ICE, this stored information can later lead to the box (and its image) being included in the invariant, or it can lead to the box being excluded if the image is excluded (or if part of the image is excluded and the box is too small to split). It can also happen that a box is split—an operation that has no analogue in ICE. The relative-completeness of our algorithm depends on using this information to delete a box only when necessary. Our algorithm also differs in the sense that it considers a set of boxes $\mathfrak{S}$ whose concretization only grows smaller over time; it works by iteratively strengthening an invariant until an inductive invariant is found. In contrast, an ICE-learner might successively consider larger or smaller candidate invariants.

Bradley presents an algorithm called ic3 [5], which synthesizes loop invariants in a property-directed manner by incrementally refining a sequence of formulas that overapproximate the program states reachable after different numbers of iterations of the loop. An ic3-like approach could be applied to the synthesis of loop invariants for the kind of programs that we investigated. However, the performance of this approach would depend on the precision and efficiency of the underlying SMT solver for (sometimes non-linear) floating-point arithmetic queries. Our algorithm has a performance advantage because it can test whether one area of the program's state space maps into another by using abstract interpretation, instead of calling an SMT solver.

To exclude unreachable states efficiently, an ic3 implementation needs to generalize from concrete counterexamples to some larger set of states; otherwise, it might only exclude one concrete state at a time. The generalization method determines the abstract domain of invariants that is being searched.

Our "quadtree" abstract domain C_Q (see Sec. 5.3) could be used to generalize each counterexample p to the largest box from Q that (i) contains p and (ii) can be soundly excluded from the invariant.

Two recent works [30,14] search for inductive invariants using abstract interpretation, and use SMT solvers to help ease the burden of designing sound abstract transformers. Both works need to represent the program's transition relation in a logic supported by the underlying SMT solver. Both also construct their inductive invariants by iteratively weakening a formula until it is inductive. In contrast, the algorithm described in this paper constructs its inductive invariants by iterative strengthening.

The algorithm presented in this paper can be compared to the large literature on abstraction refinement [9,27]. It is similar to that work, in that splitting a box can be likened to the introduction of a new predicate that distinguishes the cases where a program variable is above or below some value. Note, however, that the algorithm described in this paper works in one continuous process; the analysis does not have to be restarted when a new split is introduced. Note also that the splitting of boxes has a useful locality property: splitting one box at some value does not automatically split other boxes at the same value; thus, the new predicates are introduced locally, and on demand. The algorithm in this paper differs from counterexample-guided abstraction refinement [6] in that it makes no direct use of concrete counterexamples.

Constraint programming is a popular paradigm, with numerous applications (scheduling, packing, layout design, etc.). It has also been applied to program analysis, and in particular invariant inference, where off-the-shelf linear and nonlinear solvers [7,28], SAT solvers [32], and SMT solvers [16] have been employed. In those works, a pre-processing, or abstraction, step is necessary to transform the problem of inferring an inductive invariant, which is inherently second-order, into a first-order formula.

Our method combines abstract interpretation and constraint solving, a connection that has also been made by others. Apt observed that applying propagators can be seen as an iterative fixpoint computation [2]. Pelleau et al. expanded on this connection to describe a constraint-solving algorithm in terms of abstract interpretation; they exploited the connection to design a new class of solvers parameterized by abstract domains [24], thereby importing abstract-interpretation results and know-how into constraint programming. This paper goes the other way, and imports technology from constraint programming into abstract interpretation to help solve program-analysis problems. Several previous works, e.g., [25], combine abstract interpretation and constraint-based techniques, but generally delegate the problem of inductive-invariant inference to classic abstract-interpretation methods. Our work differs in that we devised a new algorithm for inferring inductive invariants, not a new use of an existing algorithm.

8 Conclusion

In this paper, we were inspired by one class of constraint-solving algorithms, namely, continuous constraint solving, and adapted the methods employed for such problems to devise a new algorithm for inferring inductive invariants. In-

stead of classical increasing iterations, as ordinarily used in abstract interpretation, our algorithm employs refinement (decreasing) iterations; it tightens and splits a collection of abstract elements until an inductive invariant is found. The algorithm is parameterized on an arbitrary numeric abstract domain, in which the semantics of the loop body is evaluated. The method is sound and can be implemented efficiently using floating-point arithmetic. We have shown the effectiveness of our method on small but intricate loop analyses. In particular, loops that do not have any linear inductive invariant, and thus would traditionally require the design of a novel, specialized abstract domain, have been successfully analyzed by our method, employing the interval domain only.

References

1. A. Adjé, S. Gaubert, and E. Goubault. Coupling policy iteration with semi-definite relaxation to compute accurate numerical invariants in static analysis. In *PLS'10*, volume 6012 of *LNCS*, pages 23–42. Springer, 2010.
2. K. R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221, 1999.
3. F. Benhamou, F. Goualard, L. Granvilliers, and J.-F. Puget. Revisiting hull and box consistency. In *ICLP'99*, pages 230–244, 1999.
4. 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 in AIAA, pages 1–38. AIAA, 2010.
5. A. Bradley. Sat-based model checking without unrolling. In *VMCAI'11*, pages 70–87, 2011.
6. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV'00*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.
7. M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *CAV'03*, volume 2725 of *LNCS*, pages 420–432. Springer, 2003.
8. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*, pages 238–252. ACM, Jan. 1977.
9. S. Das, D. Dill, and S. Park. Experience with predicate abstraction. In *CAV'99*, pages 160–171. Springer, 1999.
10. V. D'Silva, L. Haller, and D. Kroening. Satisfiability solvers are static analysers. In *SAS'12*, volume 7460 of *LNCS*, pages 317–333. Springer, 2012.
11. V. D'Silva, L. Haller, D. Kroening, and M. Tautschnig. Numeric bounds analysis with conflict-driven learning. In *TACAS'12*, volume 7214 of *LNCS*, pages 48–63. Springer, 2012.
12. J. Feret. Static analysis of digital filters. In *ESOP'04*, volume 2986 of *LNCS*, pages 33–48. Springer, Mar. 2004.
13. P. Garg, C. Löding, P. Madhusudan, and D. Neider. ICE: A robust framework for learning invariants. In *CAV'14*, pages 69–87. Springer, 2014.
14. P.-L. Garoche, T. Kahsai, and C. Tinelli. Incremental invariant generation using logic-based automatic abstract transformers. In *NFM'13*, 2013.
15. P. Granger. Improving the results of static analyses of programs by local decreasing iterations. In *FSTTCS'92*, 1992.
16. S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In *PLDI'08*, pages 281–292. ACM, 2008.

17. S. Itzhaky, N. Bjørner, T. Reps, M. Sagiv, and A. Thakur. Property-directed shape analysis. In *CAV'14*, pages 35–51. Springer, 2014.
18. B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV'09*, volume 5643 of *LNCS*, pages 661–667. Springer, June 2009.
19. A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In *ESOP'04*, volume 2986 of *LNCS*, pages 3–17. Springer, Mar. 2004.
20. A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
21. A. Miné, J. Breck, and T. Reps. An algorithm inspired by constraint solvers to infer inductive invariants in numeric programs. TR 1829, CS Dept., Univ. of Wisconsin, Madison, WI, January 2016.
22. U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, 7(2):95–132, 1974.
23. R. E. Moore. *Interval Analysis*. Prentice Hall, Englewood Cliffs N. J., USA, 1966.
24. M. Pelleau, A. Miné, C. Truchet, and F. Benhamou. A constraint solver based on abstract domains. In *VMCAI'13*, LNCS, page 17. Springer, Jan. 2013.
25. O. Ponsini, C. Michel, and M. Rueher. Combining constraint programming and abstract interpretation for value analysis of floating-point programs. In *CSTVA'12*, pages 775–776, 2012.
26. P. Roux and P.-L. Garoche. Practical policy iterations - A practical use of policy iterations for static analysis: the quadratic case. *FMSD*, 46(2):163–196, 2015.
27. H. Saïdi and N. Shankar. Abstract and model check while you prove. In *CAV'99*, pages 443–454. Springer, 1999.
28. S. Sankaranarayanan, H. Sipma, and Z. Manna. Constraint-based linear-relations analysis. In *SAS'04*, volume 3148 of *LNCS*, pages 53–68. Springer, 2004.
29. A. Tarski. A lattice theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–310, 1955.
30. A. Thakur, A. Lal, J. Lim, and T. Reps. PostHat and all that: Automating abstract interpretation. *ENTCS*, 311:15–32, 2015.
31. A. Thakur and T. Reps. A generalization of Stålmarck's method. In *SAS'12*, volume 7460 of *LNCS*, pages 334–351. Springer, 2012.
32. Y. Xie and A. Aiken. Saturn: A SAT-based tool for bug detection. In *CAV'05*, LNCS, pages 139–143, Berlin, Heidelberg, 2005. Springer.