

Combining Forward and Backward Abstract Interpretation of Horn Clauses

Alexey Bakhirkin^{1,2}(✉) and David Monniaux^{1,2}

¹ Univ. Grenoble Alpes, VERIMAG, 38000 Grenoble, France
{alexey.bakhirkin,david.monniaux}@univ-grenoble-alpes.fr
² CNRS, VERIMAG, 38000 Grenoble, France

Abstract. Alternation of forward and backward analyses is a standard technique in abstract interpretation of *programs*, which is in particular useful when we wish to prove unreachability of some undesired program states. The current state-of-the-art technique for combining forward (bottom-up, in logic programming terms) and backward (top-down) abstract interpretation of *Horn clauses* is query-answer transformation. It transforms a system of Horn clauses, such that standard forward analysis can propagate constraints both forward, and backward from a goal. Query-answer transformation is effective, but has issues that we wish to address. For that, we introduce a new backward collecting semantics, which is suitable for alternating forward and backward abstract interpretation of Horn clauses. We show how the alternation can be used to prove unreachability of the goal and how every subsequent run of an analysis yields a refined model of the system. Experimentally, we observe that combining forward and backward analyses is important for analysing systems that encode questions about reachability in C programs. In particular, the combination that follows our new semantics improves the precision of our own abstract interpreter, including when compared to a forward analysis of a query-answer-transformed system.

1 Introduction

In the past years, there has been much interest in using Horn clauses for program analysis, i.e., to encode the program semantics and the analysis questions as a system of Horn clauses and then use a dedicated Horn clause solver to find a model of the system or show its unsatisfiability (see e.g., [11]). In particular, collecting semantics of programs and reachability questions can be encoded as *constrained Horn clauses*, or *CHCs*.

With this approach, Horn clauses become a common language that allows different tools to exchange program models, analysis questions and analysis results. For example, as part of this work, we implemented a polyhedra-based abstract

This work was partially supported by the [European Research Council](#) under the European Union’s Seventh Framework Programme (FP/2007-2013)/ERC Grant Agreement nr. 306595 “[STATOR](#)”.

interpreter for CHCs. We use an existing tool SeaHorn [22] to convert questions about reachability in C programs into systems of CHCs, and this way we can use our abstract interpreter to analyse numeric C programs without having to ourselves implement the semantics of C. Additionally, Horn clauses allow to build complicated abstract models of programs, as opposed to implementing the abstraction mostly as part of the abstract domain. For example, D. Monniaux and L. Gonnord propose [36] a way to abstract programs that use arrays into array-free Horn clauses, and we are not aware of a domain that implements their abstraction.

On the other hand, this approach makes it more important to implement different precision-related techniques and heuristics in the analyser, since we have little control over how the problem description is formulated, when it is produced by an external procedure. One technique that is important for disproving reachability using abstract interpretation is the combination of forward and backward analyses. The idea is to alternate forward and backward analyses, and build an over-approximation of the set of states that are both reachable from the program entry and can reach an undesired state (Patrick and Radhia Cousot give a good explanation of the technique [18, section 4]).

Patrick and Radhia Cousot also propose to use a combination of forward and backward analyses a for logic programs [17]. Their combination is based on the intersection of forward (bottom-up, in logic programming terms¹) and backward (top-down) collecting semantics, which, as we observe in Sect. 3, is too over-approximate for our purposes. The current state-of-the-art technique for combining forward and backward analyses of Horn clauses is query-answer transformation [28]. The idea is to transform a system of Horn clauses, such that standard forward analysis can propagate constraints both forward from the facts, and backward from a goal. Query-answer transformation is effective, e.g., B. Kafle and J.P. Gallagher report [28] that it increases the number of benchmark programs that can be proven safe both by their abstract interpreter and by a pre-existing CEGAR-based analyser. Still, query-answer transformation has some issues, which we outline (together with its advantages) in Sect. 2.3 and revisit in Sect. 3.2.

To address the issues of the existing techniques, we introduce a new backward collecting semantics of CHCs, which offers more precision when combining forward and backward abstract interpretation. We show how the analysis based on the new semantics can be used to prove unreachability of a goal and how every subsequent run of the analysis yields a refined model of the system. In particular, if the goal is proven to be unreachable, our analysis can produce a model of the system that is disjoint from the goal, which allows to check the results of the analysis and to communicate them to other tools. These are the

¹ In this paper, we use the terms *bottom-up* and *top-down* in the meanings that they bear in logic programming and thus they correspond to *forward* and *backward* analysis respectively. In program analysis, *bottom-up* may mean *from callees to callers* or *from children to parents in the AST*, but this is *not* the meaning that we intend in this paper.

main contributions of this paper. To evaluate our approach, we take programs from the categories “loops”, and “recursive” of the Competition on Software Verification SV-COMP [2]. We use the existing tool SeaHorn to translate these programs to systems of Horn clauses. We observe that the alternation of forward and backward analyses following our new semantics improves the precision of our own abstract interpreter (i.e., it allows to prove safety of more safe programs) including when compared to forward analysis of a query-answer-transformed system.

2 Background

We say that a *term* is a variable, a constant, or an application of an *interpreted* function to a vector of terms. To denote vectors of terms, we use bold letters. Thus, \mathbf{t} denotes a vector of terms; $\varphi[\mathbf{x}]$ (assuming elements of \mathbf{x} are distinct) denotes a formula φ , where the set of free variables is the set of elements of \mathbf{x} ; and $\varphi[\mathbf{x}/\mathbf{t}]$ denotes a formula that is obtained from φ by simultaneously replacing (substituting) every occurrence of $x_i \in \mathbf{x}$ with the corresponding element $t_i \in \mathbf{t}$.

CHCs. A constrained Horn clause (CHC) is a first order formula of the form

$$\forall X. (p_1(\mathbf{t}_1) \wedge p_2(\mathbf{t}_2) \wedge \cdots \wedge p_n(\mathbf{t}_n) \wedge \varphi \Rightarrow p_{n+1}(\mathbf{t}_{n+1}))$$

where p_i are uninterpreted predicate symbols, \mathbf{t}_i are vectors of terms; φ is a quantifier-free formula in some background theory and does not contain uninterpreted predicates or uninterpreted functions; and X includes all free variables of the formula under the quantifier. Following standard notation in the literature, we write a Horn clause as

$$p_{n+1}(\mathbf{t}_{n+1}) \leftarrow \varphi, p_1(\mathbf{t}_1), p_2(\mathbf{t}_2), \cdots, p_n(\mathbf{t}_n)$$

that is, with free variables being *implicitly* universally quantified. We use a capital letter to denote an *application* of a predicate to *some* vector of terms (while for predicate symbols, we use lowercase letters). Thus, when the terms in predicate applications are not important, we can write the above clause as

$$P_{n+1} \leftarrow \varphi, P_1, P_2, \cdots, P_n$$

The predicate application P_{n+1} is called the *head* of the clause, and the conjunction $\varphi, P_1, P_2, \cdots, P_n$ is called the *body*. A CHC always has a predicate application as its head. *But*, we assume that there exists a distinguished 0-ary predicate \mathbf{f} that denotes falsity and is only allowed to appear in the head of a clause. A clause that has \mathbf{f} as its head is called an *integrity constraint*. For example, an assertion $\psi \leftarrow \varphi, P$ can be written as the integrity constraint: $\mathbf{f} \leftarrow (\varphi \wedge \neg\psi), P$.

A *system* is a set of CHCs that is interpreted as their conjunction.

Models of CHCs. We say that an *atom* is a formula of the form $p(c_1, \cdots, c_n)$, where p is an n -ary predicate symbol and c_i are constants. We denote the set of all atoms by \mathbb{A} .

An *interpretation* is a set of *atoms* $M \subseteq \mathbb{A}$. One can say that an interpretation gives *truth assignment* to atoms: an atom is interpreted as *true* if it belongs to the interpretation and as *false* otherwise. This way, an interpretation also provides a truth assignment to every formula, by induction on the formula structure.

For a system of CHCs, a *model* (or *solution*) is an interpretation that makes every clause in the system *true* (note that all variables in a system of Horn clauses are universally quantified, and thus the model does not include variable valuations). We call a model $M \subseteq \mathbb{A}$ *safe* when $\mathfrak{f} \notin M$ (many authors prefer to call an interpretation M a model only when it does not include \mathfrak{f} , but we prefer to have both notions). A system of CHCs always has the *minimal* model w.r.t. subset ordering (see, e.g., [26, section 4]). If a system has no clauses of the form $P \leftarrow \varphi$, its least model is \emptyset . We call a system of CHCs *safe* iff it has a safe model. In particular, for a safe system, its least model is safe, and thus, for a safe system, there exists the smallest safe model. For every system of CHCs, the set of atoms \mathbb{A} is the greatest (unsafe) model, but a safe system in general may not have the greatest safe model.

Fixed Point Characterization of the Least Model. A system of CHCs \mathbb{H} induces the *direct consequence relation* $\mathbb{T}_{\mathbb{H}} \subseteq \mathcal{P}(\mathbb{A}) \times \mathbb{A}$, which is constructed as follows. A tuple $(\{p_1(\mathbf{c}_1), \dots, p_n(\mathbf{c}_n)\}, p_{n+1}(\mathbf{c}_{n+1})) \in \mathbb{T}_{\mathbb{H}}$ iff the system \mathbb{H} contains a clause $p_{n+1}(\mathbf{t}_{n+1}) \leftarrow \varphi, p_1(\mathbf{t}_1), \dots, p_n(\mathbf{t}_n)$, such that $\varphi \wedge \bigwedge_{i=1}^{n+1} \mathbf{c}_i = \mathbf{t}_i$ is satisfiable.² In particular, every clause of the form $p(\mathbf{t}) \leftarrow \varphi$ induces a set of *initial transitions* (or *initial consecutions*) of the form $(\emptyset, p(\mathbf{c}))$, where $\varphi \wedge (\mathbf{c} = \mathbf{t})$ is satisfiable. Direct consequence relation can be seen as a variant of a direct consequence function discussed by J. Jaffar and M.J. Maher [26, section 4].

Note that $\mathbb{T}_{\mathbb{H}}$ is unlike an ordinary transition relation and relates a *set* of atoms with a single atom that is their direct consequence. To work with such a relation, we can adapt the standard in program analysis definition of post-condition as follows:

$$\text{post}(\mathbb{T}_{\mathbb{H}}, X) = \{a' \mid \exists A \subseteq X. (A, a') \in \mathbb{T}_{\mathbb{H}}\}$$

Then, the least model of \mathbb{H} can be characterised as the least fixed point:

$$\text{lfp}_{\subseteq} \lambda X. \text{post}(\mathbb{T}_{\mathbb{H}}, X) \tag{1}$$

As standard in abstract interpretation, we call the fixed point (1) the forward (bottom-up, in logic programming terms) collecting semantics of \mathbb{H} . In general, every pre-fixpoint of the consequence operator, i.e., every set M , s.t. $\text{post}(\mathbb{T}_{\mathbb{H}}, M) \subseteq M$ is a model of \mathbb{H} .

Analysis Questions. Given a system of CHCs \mathbb{H} , the analysis question may be stated in a number of ways. Often we want to know whether the system is safe, i.e., whether the least model of \mathbb{H} contains \mathfrak{f} . More generally, we may be

² There may be a slight abuse of notation here. When writing down the set as $\{p_1(\mathbf{c}_1), \dots, p_n(\mathbf{c}_n)\}$, we do *not* assume that all p_i or all \mathbf{c}_i are distinct and that the set has exactly n elements.

given a set of goal atoms $A_g \subseteq \mathbb{A}$. Then, the analysis question will be whether the goal is unreachable, i.e. whether the goal and the least model are disjoint. In this case, we start by computing a (reasonably small) model M of \mathbb{H} . If $M \cap A_g = \emptyset$, we conclude that the goal is unreachable. Otherwise, we either report an inconclusive result (since the computed M will in general not be the smallest model), or attempt to compute a refined model $M' \subset M$.

Alternatively, we may want to produce a model of \mathbb{H} that gives us some non-trivial information about the object described by \mathbb{H} . In this case, we usually want to produce some reasonably small model, which is what abstract interpretation tries to do. The goal may or may not be given. For example, we may be only interested in some part of the object (say, a subset of procedures in a program), which is described by a subset of predicates Π . Then, the goal will be the corresponding set of atoms $A_g = \{p(\mathbf{c}) \mid p \in \Pi\}$.

2.1 Abstract Interpretation of CHCs

Abstract interpretation [15] provides us a way to *compute an over-approximation* of the least model, following the fixed point characterization. To do so, we introduce the *abstract domain* \mathbb{D} with the least element \perp , greatest element \top , partial order \sqsubseteq and join \sqcup . Every element of the abstract domain $d \in \mathbb{D}$ represents the set of atoms $\gamma(d) \subseteq \mathbb{A}$. Then, we introduce the abstract consequence operator post^\sharp which over-approximates the concrete operator post , i.e., for every $d \in \mathbb{D}$, $\gamma(\text{post}^\sharp(H, d)) \supseteq \text{post}(\mathbb{T}_{\mathbb{H}}, \gamma(d))$. If we are able to find such element $d_m \in \mathbb{D}$ that $\text{post}^\sharp(H, d_m) \sqsubseteq d_m$ then $\gamma(d_m)$ is a pre-fixpoint of the direct consequence operator and thus a model of \mathbb{H} (not necessarily the smallest one). At this point, it does not matter how we compute d_m . It may be a limit of a Kleene-like iteration sequence (as in our implementation) or it may be produced by policy iteration [20,31], etc.

One can expect that an element $d \in \mathbb{D}$ is partitioned by predicate, in the same way as in program analysis, domain elements are partitioned by program location. In the simple case, every element $d \in \mathbb{D}$ will have a logical representation in some theory and one can think that it maps every predicate p_i to a quantifier-free formula $\delta_i[\mathbf{x}_i]$, where \mathbf{x}_i correspond to the arguments of p_i . For example, when using a polyhedral domain, d will map every predicate to a conjunction of linear constraints. For simplicity of syntactic manipulations, we can assume that \mathbf{x}_i are distinct vectors of distinct variables, i.e., a given variable appears only in one vector \mathbf{x}_i and only once.

From this, we can derive a recipe for Kleene-like iteration. Let $d \in \mathbb{D}$ be the current fixpoint candidate that maps every predicate p_i to a formula $\delta_i[\mathbf{x}_i]$. We try to find a clause $p_{n+1}(\mathbf{t}_{n+1}) \leftarrow \varphi, p_1(\mathbf{t}_1), \dots, p_n(\mathbf{t}_n)$ (where $n \geq 0$), such that the following formula is satisfiable:

$$\varphi \wedge \delta_1[\mathbf{x}_1/\mathbf{t}_1] \wedge \dots \wedge \delta_n[\mathbf{x}_n/\mathbf{t}_n] \wedge \neg \delta_{n+1}[\mathbf{x}_{n+1}/\mathbf{t}_{n+1}] \quad (2)$$

If it is, we find a *set* of models of (2), and if some model assigns the vector of constants \mathbf{c}_{n+1} to the variables \mathbf{x}_{n+1} , we join the atom $p_{n+1}(\mathbf{c}_{n+1})$ to d .

In a polyhedral analysis, we usually want to find in every step a *convex* set models of (2). Assuming the formula is in negation normal form, there is a naïve way to generalize a single model to a convex set of models by recursively traversing the formula and collecting atomic propositions satisfied by the model (descending into all sub-formulas for a conjunction and into one sub-formula for a disjunction). In general though, this corresponds to a problem of finding a model of a Boolean formula that is in some sense optimal (see, e.g., the work of J. Marques-Silva et al. [34]). When the set of CHCs is produced from a program by means of large block encoding [10] (e.g., SeaHorn does this by default), then φ is disjunctive and represents some set of paths through the original program. Finding a convex set of models of (2) corresponds to finding a path through the original program, along which we need to propagate the post-condition. In program analysis, a similar technique is called *path focusing* [23, 35].

Checking the Model. Given an element $d \in \mathbb{D}$, we can check whether it represents a model by taking its abstract consequence. If $\text{post}^\sharp(\mathbb{H}, d) \sqsubseteq d$ then $\gamma(d)$ is a pre-fixpoint of the direct consequence operator and thus is a model of \mathbb{H} . When d can be represented in a logical form and maps every predicate p_i to a formula $\delta_i[\mathbf{x}_i]$ in some theory, we can check whether it represents a model (i.e., that for every clause, the formula (2) is unsatisfiable) using an SMT solver. Being able to check the obtained models provides a building block for making a *verifiable* static analyser.

2.2 Program Analysis and CHCs

Different flavours of Horn clauses can be used to encode in logic form different program analysis questions. In particular, CHCs can be used to encode invariant generation and reachability problems. In such an encoding, uninterpreted predicates typically denote sets of reachable memory states at different program locations, clauses of the form $P_{n+1} \leftarrow \varphi, P_1, P_2, \dots, P_n$ encode the semantics of transitions between the locations, clauses of the form $P \leftarrow \varphi$ encode the initial states, and the integrity constraints (of the form $f \leftarrow \varphi, P$) encode the assertions. In this paper, we limit ourselves to invariant generation and reachability, but other program analysis questions (including verification of temporal properties [9]) can be encoded using other flavours of Horn clauses. For more information, an interested reader can refer to a recent survey [11].

Example 1 - Parallel Increment. Consider a program in Fig. 1. It starts by setting two variables, x and y , to zero and then increments both of them in a loop a non-deterministic number of times. An analyser is supposed to prove that after the loop finishes, x and y have equal values. This program also has an unreachable condition $x < 0$ upon which only x is incremented, which will be useful in the next example. The program in Fig. 1 can be encoded into CHCs as shown in Fig. 2, where the predicate p denotes the set of reachable states at the head of the loop, and its arguments denote the variables x and y respectively. From the point of view of abstract interpretation, such a system of CHCs represents a program’s collecting semantics. For simple programs, as the one in Fig. 1,

```

1 | x = y = 0;
2 | while (*) {
3 |   if (x ≥ 0) {
4 |     x += 1; y += 1;
5 |   } else {
6 |     x += 1;
7 |   }
8 | }
9 | assert(x == y);

```

Fig. 1. A program that increments x and y in parallel.

```

 $p(x, y) \leftarrow x = 0 \wedge y = 0$ 
 $p(x + 1, y + 1) \leftarrow x \geq 0, p(x, y)$ 
 $p(x + 1, y) \leftarrow x < 0, p(x, y)$ 
 $\dagger \leftarrow x \neq y, p(x, y)$ 

```

Fig. 2. Horn clause encoding of the program in Fig. 1.

a model of the system of CHCs directly represents an inductive invariant of the program. For the more complicated programs (e.g., programs with procedures) this may no longer be true, but in any case, if we find a safe (not containing f) model of the system of CHCs, we can usually conclude that the program cannot reach an assertion violation. A model that we find with abstract interpretation will assign to every predicate an element of some abstract domain; for a numeric program this may be a convex polyhedron (or a small number of polyhedra) in a space where every dimension corresponds to a predicate argument. Thus, for us to be able to prove safety of a program, the system of CHCs has to have a safe model of the given form.

Horn clause encoding of programs without procedures is typically straightforward and results in a system, where every clause has at most one predicate application in the body; such clauses are often called *linear*. Encoding of programs with procedures is also possible, but there are multiple ways of doing it. We now give an example of a program with a procedure.

```

1 | void inc_xy() {
2 |   if (x ≥ 0) {
3 |     x += 1; y += 1;
4 |   } else {
5 |     x += 1;
6 |   }
7 | }
8 | ...
9 | x = y = 0;
10 | while (*)
11 |   inc_xy();
12 | assert(x == y);

```

Fig. 3. A program that increments x and y in parallel using a procedure.

```

 $p(x, y) \leftarrow x = 0 \wedge y = 0$ 
 $p(x', y') \leftarrow f(x, y, x', y'), p(x, y)$ 
 $\dagger \leftarrow x \neq y, p(x, y)$ 
 $f(x, y, x + 1, y + 1) \leftarrow x \geq 0, f_c(x, y)$ 
 $f(x, y, x + 1, y) \leftarrow x < 0, f_c(x, y)$ 
 $f_c(x, y) \leftarrow true$ 

```

Fig. 4. A possible Horn clause encoding of the program in Fig. 3.

Example 2 - Parallel Increment Using a Procedure. Consider a program in Fig. 3. Similarly to Example 1, it starts by setting two variables, x and y , to zero and then increments both of them in a loop, but this time by calling an auxiliary procedure. Again the procedure has an unreachable condition $x < 0$ upon which it only increments x . If we encode this program into CHCs directly (without inlining of `inc_xy`), we may arrive at a system as in Fig. 4. This roughly corresponds to how the tool SeaHorn encodes procedures that do not contain assertions. As before, the predicate p denotes the reachable states at the loop head. A new predicate f denotes the *input-output relation* of the procedure `inc_xy`. If $f(x_1, y_1, x_2, y_2)$ holds, this means that if at the entry of `inc_xy` $x = x_1$ and $y = y_1$ then at the exit of `inc_xy`, it *may* be the case that $x = x_2$ and $y = y_2$. In general, every predicate that corresponds to a location inside a procedure, will have two sets of arguments: one set will correspond to the state at the entry of the procedure (as the first two arguments of f) and the other, to the corresponding state at the given location (as the last two arguments of f). Note that another new predicate, f_c , is purely auxiliary and does *not* denote the reachable states at the at the initial location of `inc_xy`. To solve the system in Fig. 4, we need to approximate the full transition relation of `inc_xy`, which includes approximating the outputs for the inputs, with which the procedure is never called. If we analyse this program in a polyhedral domain, we will notice that the full input-output relation of `inc_xy` cannot be approximated in a useful way by a single convex polyhedron. But if we restrict the analysis to the reachable states, where $x \geq 0$ always holds, we will be able to infer that `inc_xy` increments both x and y , and this will allow to prove safety of the program.

One may argue that we should alter the way we encode procedures and constrain f_c to denote the set of reachable states at the entry of `inc_xy`. But when building an analysis tool, we should cater for different possible encodings.

2.3 Combination of Forward and Backward Program Analyses

Example 2 demonstrates the general problem of communicating analysis results between different program locations. In an inter-procedural analysis, often we do not want to explicitly build the full input-output relation of a procedure. For the inputs, with which a procedure may be called, we *do* want to find the corresponding outputs, but for the other inputs we may want to report that the output is unknown. This is because often, as in Example 2, the full input-output relation will not have a useful approximation as a domain element. At the same time, a useful approximation may exist when we consider only reachable inputs. Similar considerations hold for intra-procedural analysis. If we want to prove that an assertion violation is unreachable, we do not need to explicitly represent the full inductive invariant of a program. Instead, we want to approximate the set of states that are both reachable from the initial states and may reach an assertion violation. If this set turns out to be empty, we can conclude that an assertion violation is unreachable. This technique is standard for program analysis, and in Sect. 3, we adapt it to CHCs.

An alternative technique for Horn clauses is query-answer transformation [28]. Given the original system of CHCs \mathbb{H} , we build the transformed system \mathbb{H}^{qa} . For every uninterpreted predicate p in \mathbb{H} (including f), \mathbb{H}^{qa} contains a query predicate p^q and an answer predicate p^a . The clauses of \mathbb{H}^{qa} are constructed as follows.

- *Answer clauses.* For every clause $P_{n+1} \leftarrow \varphi, P_1, \dots, P_n$ (where $n \geq 0$) in \mathbb{H} , the system \mathbb{H}^{qa} contains the clause $P_{n+1}^a \leftarrow \varphi, P_{n+1}^q, P_1^a, \dots, P_n^a$.
- *Query clauses.* For every clause $P_{n+1} \leftarrow \varphi, P_1, \dots, P_n$ (where $n \geq 0$) in \mathbb{H} , the system \mathbb{H}^{qa} contains the clauses:

$$\begin{aligned} P_1^q &\leftarrow \varphi, P_{n+1}^q \\ P_2^q &\leftarrow \varphi, P_{n+1}^q, P_1^a \\ &\dots \\ P_n^q &\leftarrow \varphi, P_{n+1}^q, P_1^a, \dots, P_{n-1}^a \end{aligned}$$

- *Goal clause* $\text{f}^q \leftarrow \text{true}$.

Then, forward (bottom-up) analysis of \mathbb{H}^{qa} corresponds to a combination of forward and backward (top-down) analyses of \mathbb{H} .

We experienced several issues with the query-answer transformation. For linear systems of CHCs, forward analysis of \mathbb{H}^{qa} corresponds to a single run of backward analysis of \mathbb{H} followed by a single run of forward analysis. For non-linear systems, this gets more complicated, though, as there will be recursive dependencies between query and answer predicates, and the propagation of information will depend on the order, in which query clauses are created. We observed that is not enough, and for some systems the analysis needs to propagate the information forward and then backward multiple times. This usually happens when the abstract domain of the analysis cannot capture the relation between the program variables.

Example 3. In Fig. 5, we show a synthetic example of a program that needs more than one alternation of forward and backward analysis to be proven safe. Notice that this program is safe, as after entering the `if`-branch in line 4 we have that $x > 0$ and $x = ky$ for some $k \geq 0$, therefore y is also greater than 0, and this is not changed by adding x to y in lines 5–6. If we work in a polyhedral domain, we cannot capture the relation $\exists k \geq 0. x = ky$ and therefore should proceed with the safety proof in a different way, e.g., as follows. First, we run a forward analysis and establish that at lines 5–7, $x > 0$, since these lines are inside the `if`-branch. Then, we run a backward analysis starting with the set of states $y < 0$ at line 7, which corresponds to the assertion violation. Since the loop in lines 5–6 can only increase y , we establish that for y to be less than zero in line 7, it also has to be less than zero in lines 1–6. Finally, we run forward analysis again and establish that for the assertion violation to be reachable, x at line 4 has to be both greater than zero (so that we enter the `if`-branch), and less-or-equal to zero (because x starts being zero and in lines 2–3 we repeatedly add a negative number to it), which is not possible. While this particular example is synthetic, in our experiments we observe a small number of SV-COMP programs where a similar situation arises.

```

1 | x = 0; y = *;
2 | while (*)
3 |   x += y;
4 | if (x > 0) {
5 |   while (*)
6 |     y += x;
7 |   assert(y ≥ 0);
8 | }

```

Fig. 5. Program, where polyhedral analysis needs to propagate information forward and backward multiple times.

A more subtle (but more benign) issue is that when solving the query-answer-transformed system, we are actually not interested in the elements of the interpretation of p^a , which are outside of p^q , but this is not captured in \mathbb{H}^{qa} itself. Because of this, p^a may be over-approximated too much as a result of widening or join. Perhaps this is one of the reasons why B. Kafle and J.P. Gallagher propose [28] to perform abstract interpretation in two phases. First, they analyse the transformed system \mathbb{H}^{qa} . Then, they strengthen the original system with the interpretations of *answer* predicates and run an analysis on the strengthened system.

To address these issues, we decided to adapt the standard (for program analysis) alternation of forward and backward analysis to CHCs. We return to the comparison of our approach to query-answer transformation in Sect. 3.2.

3 Combining Forward and Backward Analysis of CHCs

Patrick and Radhia Cousot proposed a backward (top-down) semantics for Horn clauses, which collects atoms that can appear in an SLD-resolution proof [17]. We take their definition as a starting point and define a new backward semantics and a new more precise combined forward-backward semantics. Then we show, how we can use our new semantics to disprove reachability of a goal and to refine a model w.r.t. the goal.

Backward Transformers and Collecting Semantics. First, let us introduce the pre-condition operation as follows. For a system \mathbb{H} ,

$$\text{pre}(\mathbb{T}_{\mathbb{H}}, A') = \{a \mid \exists A \subseteq \mathbb{A}. \exists a' \in A'. (A, a') \in \mathbb{T}_{\mathbb{H}} \wedge a \in A\}$$

Then, for a system \mathbb{H} and a set of goal atoms A_g , the backward (top-down) semantics is characterized by the least fixed point:

$$\text{lfp}_{\subseteq} \lambda X. A_g \cup \text{pre}(\mathbb{T}_{\mathbb{H}}, X) \tag{3}$$

which corresponds to the semantics proposed by Patrick and Radhia Cousot. This definition of backward semantics has a drawback though. The intersection of forward semantics (1) and backward semantics (3) *over-approximates* the set of atoms that can be derived from initial clauses (of the form $P \leftarrow \varphi$) and can be used to derive the goal.

Example 4. Let us consider the following system of CHCs, where p is a unary predicate and c_1, \dots, c_5 are constants

$$\begin{aligned}
p(c_1) &\leftarrow \text{true} & p(c_5) &\leftarrow p(c_3) \\
p(c_2) &\leftarrow p(c_1) & p(c_5) &\leftarrow p(c_2), p(c_4) \\
p(c_3) &\leftarrow p(c_1) & &
\end{aligned} \tag{4}$$

The forward semantics (1) for this system is the set $\{p(c_1), p(c_2), p(c_3), p(c_5)\}$ (note that the atom $p(c_4)$ cannot be derived). Let us assume that the set of goals is $A_g = \{p(c_5)\}$. Then, the backward semantics (3) for this system is $\{p(c_1), p(c_2), p(c_3), p(c_4), p(c_5)\}$. The intersection of forward and backward semantics is $\{p(c_1), p(c_2), p(c_3), p(c_5)\}$, even though the atom $p(c_2)$ is not used when deriving the goal $\{p(c_5)\}$ (because we cannot derive $p(c_4)$). If we implement an abstract analysis based on the intersection of semantics (1) and (3), this will become an additional source of imprecision.

3.1 Forward and Backward Analyses Combined

We wish to define a combination of forward and backward semantics that does not introduce the over-approximation observed in Example 4. For that, we propose the *restricted pre-condition* operation that we define as follows. For a restricting set $R \subseteq \mathbb{A}$,

$$\text{pre}|_R(\mathbb{T}_{\mathbb{H}}, A') = \{a \mid \exists A \subseteq R. \exists a' \in A'. (A, a') \in \mathbb{T}_{\mathbb{H}} \wedge a \in A\}$$

Now, we can define the combined forward-backward collecting semantics as follows:

$$\begin{aligned} & \text{lfp}_{\subseteq} \lambda X. (A_g \cap M) \cup \text{pre}|_M(\mathbb{T}_{\mathbb{H}}, X) \\ & \text{where } M = \text{lfp}_{\subseteq} \lambda X. \text{post}(\mathbb{T}_{\mathbb{H}}, X) \end{aligned} \quad (5)$$

One can show that this semantics denotes the set of atoms that can be derived from initial clauses (of the form $P \leftarrow \varphi$) and can be used to derive the goal (we defer an explanation until Sect. 5). For example, one can see that for the system (4) discussed in Example 4, computing this semantics produces the set $\{p(c_1), p(c_3), p(c_5)\}$, as expected.

Introducing a restricted pre-condition operation is common, when a combination of analyses cannot be captured by the meet operation in the domain. For example, assume that we want to analyse the instruction $z := x + y$ in an interval domain. Assume also that the *pre-condition* is restricted by $x \geq 3$ (e.g., obtained by forward analysis) and the *post-condition* is $z \in [0, 2]$. In this case, *unrestricted* backwards analysis yields no new results. But if we modify the pre-condition operation to take account of the previously obtained pre-condition ($x \geq 3$ in this case), we can derive the new constraint $y \leq -1$.

It may however be unusual to see a restricted pre-condition in concrete collecting semantics. To explain it, in Sect. 5, we introduce tree semantics of CHCs and show how concrete collecting semantics is itself an abstraction of tree semantics. In particular, the intersection of forward and backward tree semantics abstracts to (5).

Abstract Transformers. As standard in abstract interpretation, we introduce over-approximate versions of forward and backward transformers, resp. post^\sharp and pre^\sharp , s.t. for $d, r \in \mathbb{D}$,

$$\gamma(\text{post}^\sharp(\mathbb{H}, d)) \supseteq \text{post}(\mathbb{T}_{\mathbb{H}}, \gamma(d)) \quad \gamma(\text{pre}^\sharp|_r(\mathbb{H}, d)) \supseteq \text{pre}|_{\gamma(r)}(\mathbb{T}_{\mathbb{H}}, \gamma(d))$$

Abstract Iteration Sequence. In concrete world, the combination of forward and backward analyses is characterized by a pair of fixed points in (5). In particular, we have the following property:

Proposition 1. *If we let $M = \text{lfp}_{\subseteq} \lambda X. \text{post}(\mathbb{T}_{\mathbb{H}}, X)$ and $M' = \text{lfp}_{\subseteq} \lambda X. (A_g \cap M) \cup \text{pre}|_M(\mathbb{T}_{\mathbb{H}}, X)$ then $\text{lfp}_{\subseteq} \lambda X. (\text{post}(\mathbb{T}_{\mathbb{H}}, X) \cap M') = M'$.*

That is, concrete forward and backward analyses need not be iterated. We give the proof of this a bit later. In the abstract world, this is not the case, as has already been noted for program analysis [18]. In general, given the abstract goal $g \in \mathbb{D}$, the combination of abstract forward and backward analyses produces the sequence:

$$\begin{aligned} & b_0, d_1, b_1, d_2, b_2, \dots, \text{ where} \\ & b_0 = \top, \text{ and for } i \geq 1, \\ & \text{post}^\#(\mathbb{H}, d_i) \sqcap b_{i-1} \sqsubseteq d_i \\ & g \sqcap d_i \sqsubseteq b_i \\ & \text{pre}^\#|_{d_i}(\mathbb{H}, b_i) \sqsubseteq b_i \end{aligned} \tag{6}$$

In principle, this iterations sequence may be infinitely descending, and to ensure termination of an analysis, we have to limit how many elements of the sequence are computed. In our experiments though, the sequence usually stabilizes after the first few elements.

Propositions 2 and 3 respectively show how we can refine the initial model w.r.t. the goal and how we can use the iteration sequence to disprove reachability of the goal.

Proposition 2. *For every $k \geq 1$, the set $\gamma(d_k) \cup \bigcup_{i=1}^{k-1} (\gamma(d_i) \setminus \gamma(b_i))$ is a model of \mathbb{H} .*

We present the proof in Appendix A.

Observe that for some abstract domains (e.g., common numeric domains: intervals, octagons, polyhedra), the meet operation is usually exact, i.e. for $d_1, d_2 \in \mathbb{D}$, $\gamma(d_1 \sqcap d_2) = \gamma(d_1) \cap \gamma(d_2)$. Also, for such domains we can expect that for $r, d \in D$, $\text{pre}^\#|_r(\mathbb{H}, d) \sqsubseteq r$. In this case, the forward-backward iteration sequence is descending: $b_0 \supseteq d_1 \supseteq b_1 \supseteq d_2 \supseteq \dots$, and computing every subsequent element d_i provides a tighter model of \mathbb{H} (assuming d_i is distinct from d_{i-1}). This comes at a cost, though, since the refined model will not in general be expressible in the abstract domain of the analysis. For example, in a polyhedral analysis, when d_i and b_i are maps from predicates to convex polyhedra, expressing the model given by Proposition 2, requires finite sets of convex polyhedra. If we wish to check if such an object M is indeed a model of \mathbb{H} , we will need to check that M *geometrically covers* its post-condition. This can be done using a polyhedra library that supports powerset domains and geometric coverage (e.g., Parma Polyhedra Library [7]) or with an SMT-solver.

Now, the proof of Proposition 1 becomes straightforward.

Proof (of Proposition 1). Let $M'' = \text{lfp}_{\subseteq} \lambda X. (\text{post}(\mathbb{T}_{\mathbb{H}}, X) \cap M')$, i.e. $M'' \subseteq M'$ by definition. From Proposition 2, $(M \setminus M') \cup M'' \subseteq M$ is a model of \mathbb{H} . Since M is the smallest model, $(M \setminus M') \cup M'' = M$ and $M'' = M'$.

Proposition 3. *If there exists $k \geq 1$, s.t. $d_k = \perp$, then there exists a model M of \mathbb{H} , s.t. $M \cap \gamma(g) = \emptyset$ (i.e., the goal is unreachable).*

Proof. If $d_k = \perp$ then $\gamma(d_k) = \emptyset$, and from Proposition 2, $M = \bigcup_{i=1}^{k-1} (\gamma(d_i) \setminus \gamma(b_i))$ is a model of \mathbb{H} . From (6), it follows that for every i , $\gamma(g) \cap \gamma(d_i) \subseteq \gamma(b_i)$, that is $(\gamma(d_i) \setminus \gamma(b_i)) \cap \gamma(g) = \emptyset$. This means that $M \cap \gamma(g) = \emptyset$.

Thus, when there exists k s.t. $d_k = \perp$, we obtain a *constructive* proof of unreachability of the goal that can later be checked.

Result of the Analysis. Propositions 2 and 3 provide a way to give additional information to the user of the analysis, apart from the verdict (*safe* or *potentially unsafe*). Suppose, we compute the iteration sequence (6) up to the element d_k and then stop (whether because $d_k = \perp$, or the sequence stabilized, or we reached a timeout, etc.). The object d_k in itself may not be interesting: it is not a model of \mathbb{H} , it is not a proof or a refutation of reachability of the goal. If the user wishes to check the results of the analysis, we may give them the whole iteration sequence up to d_k . Then, the user will need to confirm that the sequence indeed satisfies the conditions of (6). Alternatively, we may give the user the refined model of \mathbb{H} , i.e. some representation of $M = \gamma(d_k) \cup \bigcup_{i=1}^{k-1} (\gamma(d_i) \setminus \gamma(b_i))$. This will allow the user to not only check the model, but also, e.g., produce program invariants that can be used by another verification tool (e.g., Frama-C [3], KeY [5], etc.). Representation of M may require an abstract domain that is more expressive than the domain of the analysis, but may be more compact than the whole iteration sequence. Alternatively, if d_i and b_i can be represented in logical form in some theory, so can M .

Which Analysis Runs First. In the iteration sequence (6), forward and backward analyses alternate, but which analysis runs first is actually not fixed. We may start with forward analysis and compute d_1 as normal, or we may take $d_1 = \top$ and start the computation with backward analysis. A notable option is to do the first run of backward analysis in a more coarse abstract domain and switch to a more precise domain in subsequent runs. For example, the initial run of backward analysis may only identify the predicates that can potentially be used to derive the goal:

$$\begin{aligned} & \text{lfp}_{\subseteq} \lambda X. \Pi_g \cup \text{pre}(T_{\Pi}, X), \text{ where} \\ & \Pi_g = \{p \mid p(\mathbf{c}) \in A_g\} \\ & T_{\Pi} = \{(\Pi, p') \mid \exists (A, a') \in \mathbb{T}_{\mathbb{H}}. \Pi = \{p \mid p(\mathbf{c}) \in A\} \wedge a' = p'(\mathbf{c}')\} \end{aligned} \quad (7)$$

Then, we can take $d_1 = \top$, b_1 to be some abstraction of (7), and starting from d_2 , run the analysis with a more precise domain. In program analysis, restricting

attention to program locations that have a path to (i.e., are backward-reachable from) some goal location, is a known technique. For example, K. Apinis, H. Seidl, and V. Voidani describe a sophisticated version of it [6].

3.2 Revisiting the Query-Answer Transformation

In principle, the iteration sequence (6) can be emulated by an iterated simple query-answer transformation. Let \mathbb{H} be the original system of CHCs. Let the element b_k of the iteration sequence (6) map every predicate p_i to a formula β_k^i . In particular, b_0 will map every p_i to *true*. Then, d_{k+1} can be found as a model of the system \mathbb{H}_{k+1}^d . To construct, \mathbb{H}_{k+1}^d , for every CHC $P_{n+1} \leftarrow \varphi, P_1, \dots, P_n$ (for $n \geq 0$) in the original system \mathbb{H} , we add to \mathbb{H}_{k+1}^d the clause $P_{n+1} \leftarrow \varphi \wedge \beta_k^{n+1}, P_1, \dots, P_n$. Now let the element d_k map every P_i to a formula δ_k^i . Then, b_k can be found as a model of the system \mathbb{H}_k^b that is constructed as follows. For every CHC in the original system \mathbb{H} : $P_{n+1} \leftarrow \varphi, P_1, \dots, P_n$, we add to \mathbb{H}_k^b the clauses $P_1 \leftarrow \varphi \wedge \bigwedge_{i=1}^n \delta_k^i, P_{n+1}$ through $P_n \leftarrow \varphi \wedge \bigwedge_{i=1}^n \delta_k^i, P_{n+1}$. Also, we add to \mathbb{H}_k^b the goal clause $f \leftarrow f_k$. If we compute the elements of the iteration sequence up to d_k , then the function that maps every p_i to $\delta_k^i \vee \bigvee_{j=1}^{k-1} (\delta_j^i \wedge \neg \beta_j^i)$ represents a model of the original system \mathbb{H} . In particular, when $d_1 = \top$, and $k = 2$, this produces a model, where every p_i maps to $\beta_1^i \Rightarrow \delta_2^i$.

Thus, one has a choice, whether to take a fixpoint-based approach, as we did, or a transformation-based approach. From the theoretical point of view, one will still have to prove that the iterated transformation allows to prove unreachability of the goal and to build a refined model, i.e., some analog of Propositions 2 and 3. As one can see in Appendix A, this is not trivial for the steps beyond the second. From the practical point of view, we believe that our approach allows to more easily implement some useful minor features. For example, the iteration sequence (6) naturally constrains b_i to be below d_i and d_i to be below b_{i-1} , which in some cases makes widening and join less aggressive. It should be possible though to achieve a similar effect for the query-answer transformation at the expense of introducing additional predicates and clauses.

On the other hand, an advantage of query-answer transformation is that it can be used as a preprocessing step for the analyses that are not based on abstract interpretation. For example, B. Kafle and J.P. Gallagher report [28] that it can improve the precision of a CEGAR-based analyser.

4 Implementation and Experiments

We implemented our approach in a prototype abstract interpreter. It can analyse numeric C programs that were converted to a system of CHCs with the tool SeaHorn [22] (the input format is currently a technical limitation, and we wish to remove it in the future). The implementation is written in OCaml and available online [4]. A notable feature of SeaHorn is that it introduces Boolean variables and predicate arguments even for programs without Boolean variables. To represent sets of valuations of numeric and Boolean variables, we use Bddapron [27].

We implement Kleene-like iteration as outlined in Sect. 2.1, which is similar to path focusing [23, 35]. Iteration order and choice of widening points are based on F. Bourdoncle’s [13, 14] recursive strategy (except that we implement it using a worklist with priorities). As an SMT solver, we use Z3 [12]. For comparison, in addition to the forward-backward iteration sequence (6), we implemented an analysis based on query-answer transformation.

To evaluate our implementation, we took C programs from the categories *loops* and *recursive* of the Competition on Software Verification SV-COMP [2]. SeaHorn operates on LLVM bytecode produced by Clang [1], and the resulting system of CHCs depends a lot on Clang optimization settings. For example, constant folding may remove whole computation paths when they do not depend on non-deterministic inputs. Or, Clang may replace recursion with a loop, which will make SeaHorn produce a linear system of CHCs instead of a non-linear one. In our experiments, we compiled the input programs with two optimization levels: `-O3` (SeaHorn’s default) and `-O0`. As a result, we get a total of 310 systems of Horn clauses, out of which 158 are declared safe by SV-COMP. Since we cannot prove unsafety, our evaluation focuses on safe systems. Out of 158 safe systems, our tool can work with 123. Other systems use features that are not yet supported in our tool (division, non-numeric theories, etc.). Out of 158 safe systems, 74 are non-linear.

First, we evaluate the effect of combined forward-backward analysis. The results are presented in Table 1. We compare three approaches. The first is the one we propose in this paper, i.e., based on the forward-backward iteration sequence (6). We compute the elements of (6) up to d_5 . If we decrease the limit from d_5 to d_3 , we can prove safety of 2 less programs; increasing the limit to d_7 gives no effect. The second one is a 2-step analysis based on query-answer transformation [28]. First, it runs forward analysis on a query-answer transformed system, then injects the interpretations of answer predicates in the original system and runs forward analysis again. We implemented this analysis ourselves, and thus we are *not* directly comparing our implementation to the tool Rahft [30], where this analysis was first implemented. Finally, we also run a simple forward analysis. In Table 1, we report the number of programs that we *proved safe* with every approach. One can see that our approach has a small advantage over both query-answer transformation and simple forward analysis. Interestingly, B. Kafle and J.P. Gallagher report [28] a *much* greater difference when moving from simple forward analysis to query-answer transformation. This can be attributed to three factors. First, their set of benchmarks is different, although it includes many programs from the same SV-COMP categories. Second, their benchmarks are, to our knowledge, not pre-processed by Clang. Third, as B. Kafle and J.P. Gallagher themselves report, some issues solved by adding backward analysis can as well be solved by path focusing, which our tool implements.

For reference, we also compare our tool to the solver that is integrated with SeaHorn (to our knowledge, it is based on the tool SPACER. [32, 33]). We present the results in Table 2. SeaHorn can prove safety of more programs, which is expected since our tool is in an early stage of development.

Table 1. Comparison of abstract interpretation strategies.

		Proven safe		
Safe	Supported	This paper	QA	Fwd
158	123	87	82	76

Table 2. Comparison to SeaHorn’s builtin solver (with 1 min timeout).

	This paper	SeaHorn
Proven safe	87/123 (70%)	133/158 (84%)

5 Tree Semantics of CHCs

In this section, we briefly introduce tree semantics of CHCs. Trees are not convenient objects to work with, and studying tree semantics is not the main purpose of this paper. Thus, our description will not be fully rigorous. Rather, our goal is to give the reader an intuition of why we construct collecting semantics (especially, backward and combined semantics) in the way we do, which is perhaps best explained when collecting semantics is viewed as an abstraction of tree semantics.

For the purpose of this section, a *tree* is either a leaf node containing an atom, or an interior node that contains an atom and also has a non-zero number of child subtrees.

$$\text{Tree} ::= \text{leaf}(a) \mid \text{tree}(a \leftarrow t_1, \dots, t_n)$$

where $a \in \mathbb{A}$ and every t_i is a tree. The *root atom* of a tree is naturally defined as

$$\text{root}(\text{leaf}(a)) = a \quad \text{root}(\text{tree}(a \leftarrow t_1, \dots, t_n)) = a$$

The set of *leaves* of a tree is defined as

$$\text{leaves}(\text{leaf}(a)) = \{a\} \quad \text{leaves}(\text{tree}(a \leftarrow t_1, \dots, t_n)) = \bigcup_{i=1}^n \text{leaves}(t_i)$$

The tree semantics of a system of CHCs \mathbb{H} is a set of trees, where the parent-child relation is defined by the direct consequence relation $\mathbb{T}_{\mathbb{H}}$. To get more formal, let us first define the post-condition operation on trees as follows:

$$\begin{aligned} \text{post}^t(\mathbb{H}, X) = & \{ \text{tree}(a' \leftarrow t_1, \dots, t_n) \mid t_1, \dots, t_n \in X \\ & \wedge \exists (A, a') \in \mathbb{T}_{\mathbb{H}}. |A| = n \wedge A = \{ \text{root}(t_1), \dots, \text{root}(t_n) \} \} \cup \\ & \{ \text{leaf}(a) \mid (\emptyset, a) \in \mathbb{T}_{\mathbb{H}} \} \end{aligned}$$

Intuitively, the operation performs two distinct actions: (i) it produces a trivial tree $\text{leaf}(a)$ for every initial transition (\emptyset, a) ; and (ii) for every non-initial transition (A, a') , it creates every possible tree $\text{tree}(a' \leftarrow t_1, \dots, t_n)$, where t_i are elements of X , and their roots correspond to distinct elements of A . Then, we can define the *forward tree semantics* of \mathbb{H} as the least fixed point:

$$\text{lfp}_{\subseteq} \lambda X. \text{post}^t(\mathbb{H}, X)$$

Intuitively, this is the set of trees, where leaves are initial atoms, and parent-child relation is defined by the direct consequence relation. One can say that this is the set of derivation trees induced \mathbb{H} . A notable property of forward tree semantics is that it is *subtree-closed*, i.e., with every tree, it also contains all of its subtrees.

Let us now define the *set-of-atoms* abstraction of a set of trees. First, let us define an auxiliary predicate that tells whether an atom is a node of a tree.

$$\begin{aligned} \text{isnode}(a, \text{leaf}(a')) &= (a = a') \\ \text{isnode}(a, \text{tree}(a' \leftarrow t_1, \dots, t_n)) &= (a = a') \vee \bigvee_{i=1}^n \text{isnode}(a, t_i) \end{aligned}$$

Then, for a set of trees T , its set-of-atoms abstraction is

$$\alpha^t(T) = \{a \mid \exists t \in T. \text{isnode}(a, t)\}$$

In particular, when T is subtree-closed, one can show that

$$\alpha^t(T) = \{\text{root}(t) \mid t \in T\} \quad (8)$$

Let us observe that the set-of-atoms abstraction of the forward tree semantics is exactly the forward collecting semantics:

Proposition 4. $\alpha^t(\text{lfp}_{\subseteq} \lambda X. \text{post}^t(\mathbb{H}, X)) = \text{lfp}_{\subseteq} \lambda X. \text{post}(\mathbb{T}_{\mathbb{H}}, X)$

Proof (sketch). This is an instance of *exact fixed point abstraction* [16, theorem 7.1.0.4], and to prove the proposition, we need to show that

$$\alpha^t(\text{post}^t(\mathbb{H}, T)) = \text{post}(\mathbb{T}_{\mathbb{H}}, \alpha^t(T)) \quad (9)$$

This is not true for an arbitrary T , but can be shown as true when T is subtree-closed, as it follows from (8). The post^t operation preserves subtree-closure, thus Proposition 4 can be seen as a fixed point in the lattice of subtree-closed sets, where (9) holds and thus exact fixed point abstraction holds as well.

Let us now define the *backward tree semantics*. For a set of trees T , let $\text{pre}^t(\mathbb{H}, T)$ be the set of trees that are produced from trees in T by replacing a single leaf containing $a' \in \mathbb{A}$ with a subtree $\text{tree}(a' \leftarrow a_1, \dots, a_n)$, s.t. a_1, \dots, a_n are distinct, and $(a', \{a_1, \dots, a_n\}) \in \mathbb{T}_{\mathbb{H}}$. Also let $T_g = \{\text{leaf}(a) \mid a \in A_g\}$. Then, the backward tree semantics of \mathbb{H} is the least fixed point

$$\text{lfp}_{\subseteq} \lambda X. T_g \cup \text{pre}^t(\mathbb{H}, X)$$

Intuitively, this is the set of trees where the root is in A_g , and parent-child relation is defined by the direct consequence relation.

Let us define a *pre-tree* of a tree t to be an object that is a tree and that is produced by selecting a number (possibly, zero) of non-root interior nodes and replacing every such interior node $\text{tree}(a \leftarrow t_1, \dots, t_n)$ with the leaf $\text{leaf}(a)$.

A notable property of backward tree semantics is that it is *pre-tree-closed*, i.e., with every tree, it also contains all of its pre-trees. One can show that when T is pre-tree closed,

$$\alpha^t(T) = \bigcup \{ \text{leaves}(t) \mid t \in T \}$$

Similarly to the forward case, the set-of-atoms abstraction of the backward tree semantics is exactly the backward collecting semantics.

Proposition 5. $\alpha^t(\text{lfp}_{\subseteq} \lambda X. T_g \cup \text{pre}^t(\mathbb{H}, X)) = \text{lfp}_{\subseteq} \lambda X. A_g \cup \text{pre}(\mathbb{T}_{\mathbb{H}}, X)$

Proof (sketch). The proof idea is similar to that of Proposition 4. We need to show that $\alpha^t(T_g \cup \text{pre}^t(\mathbb{H}, T)) = A_g \cup \text{pre}(\mathbb{T}_{\mathbb{H}}, \alpha^t(T))$ which does hold when T is pre-tree-closed; and pre-tree-closure is preserved by the transformer $\lambda X. T_g \cup \text{pre}^t(\mathbb{H}, X)$.

Now, let us consider the intersection of the forward and backward tree semantics: $(\text{lfp}_{\subseteq} \lambda X. \text{post}^t(\mathbb{H}, X)) \cap (\text{lfp}_{\subseteq} \lambda X. T_g \cup \text{pre}^t(\mathbb{H}, X))$. This is the set of trees that have initial atoms as leaves and a goal atom as root. We can now observe that the combined forward-backward semantics (5) is exactly the set-of-atoms abstraction of this object.

Proposition 6. $\alpha^t\left(\left(\text{lfp}_{\subseteq} \lambda X. \text{post}^t(\mathbb{H}, X)\right) \cap \left(\text{lfp}_{\subseteq} \lambda X. T_g \cup \text{pre}^t(\mathbb{H}, X)\right)\right)$
 $= \text{lfp}_{\subseteq} \lambda X. (A_g \cap M) \cup \text{pre}|_M(\mathbb{T}_{\mathbb{H}}, X)$
where $M = \text{lfp}_{\subseteq} \lambda X. \text{post}(\mathbb{T}_{\mathbb{H}}, X)$

To see intuitively why this is true, let $t \in (\text{lfp}_{\subseteq} \lambda X. \text{post}^t(\mathbb{H}, X)) \cap (\text{lfp}_{\subseteq} \lambda X. T_g \cup \text{pre}^t(\mathbb{H}, X))$ and let us observe which atoms may appear in t at different depth. We know that $\text{root}(t) \in A_g \cap M$. At depth one, we will observe sub-trees that have initial atoms as leaves and can be combined to produce t . One can see that the set of atoms at depth one is $\text{pre}|_M(\mathbb{T}_{\mathbb{H}}, A_g \cap M)$. Similarly, the set of atoms at depth two is $\text{pre}|_M(\mathbb{T}_{\mathbb{H}}, \text{pre}|_M(\mathbb{T}_{\mathbb{H}}, A_g \cap M))$. Continuing this way, we get that the set-of-atoms abstraction of the intersection of forward and backward tree semantics is $\text{lfp}_{\subseteq} \lambda X. (A_g \cap M) \cup \text{pre}|_M(\mathbb{T}_{\mathbb{H}}, X)$.

To summarise, the combined forward-backward semantics (5) is the set-of-atoms abstraction of the intersection of forward and backward tree semantics. Since set-of-trees intersection and set-of-states abstraction do not commute, we need to introduce the restricted pre-condition operation to define the combined semantics.

6 Related Work

Combining forward and backward analyses is standard when analysing programs. A good explanation of the technique is given by Patrick and Radhia Cousot [18, section 4]. They also propose to use it for the analysis of logic programs [17]. Their combination is an intersection of forward and backward collecting semantics.

F. Benoy and A. King were perhaps the first to apply abstract interpretation in a *polyhedral domain* to constraint logic programs [8]. J.P. Gallagher et al. in a series of works (see, e.g., [28,37]) apply it to specialized CLPs or CHCs. Previous sections discuss the differences between their approach and ours. Later work by B. Kafle, J.P. Gallagher, and J.F. Morales [29,30] introduces another analysis engine that is not based on abstract interpretation. M. Proietti, F. Fioravanti et al. propose a similar analysis [19] that iteratively specializes the initial system of CHCs by propagating constraints both forward and backward and by heuristically applying join and widening operators. This process is repeated until the analysis arrives at a system that can be trivially proven safe or a timeout is reached. Notably, this analysis avoids explicitly constructing the model of the original system.

Multiple researchers were advocating using Horn clauses for program verification, including A. Rybalchenko [21], N. Bjørner, and others. A survey was recently made by N. Bjørner, A. Gurfinkel, K. McMillan, and A. Rybalchenko [11]. Tools that allow to solve problems stated as systems of Horn clauses include E-HSF [9], Eldarica [39], Z3 (with PDR [25] and SPACER [32,33] engines), and others. As our implementation is in early development, we do not make a detailed comparison to these tools.

Path focusing was described by D. Monniaux and L. Gonnord [35] and implemented by J. Henry, D. Monniaux, and M. Moy in a tool PAGAI [23]. This is an approach to abstract interpretation, where one uses an SMT solver to find a path through a program, along which to propagate the post-conditions.

7 Conclusion and Future Work

In this paper, we introduce a new backward collecting semantics, which is suitable for alternating forward and backward abstract interpretation of Horn clauses. We show how the alternation can be used to prove unreachability of the goal and how every subsequent run of an analysis yields a refined model of the system. Experimentally, we observe that combining forward and backward analyses is important for analysing systems that encode questions about reachability in C programs. In particular, the combination that follows our new semantics improves the precision of our own abstract interpreter, including when compared to a forward analysis of a query-answer-transformed system.

We see the following directions for future work. *First*, we wish to be able to infer models that are *disjunctive* in a meaningful way. Currently, as we use Bddapron, we produce models where a predicate maps to a disjunctive formula, but the disjunctions are defined by the Boolean arguments of the predicate, which are often unrelated to the interesting facts about numeric arguments. We wish to explore how partitioning approaches designed for program analysis [24,38] can be applied to the analysis of Horn clauses. *Second*, we note that currently, for the combination of forward and backward analyses to work, we need to explicitly specify the goal (**query**, in terms of SeaHorn language). It would be nice though, if we could use the benefits of the combined analysis (e.g., analysing

the procedures only for reachable inputs) without having an explicit goal. For that, we will need to be able to distinguish, which of the clauses of the form $P \leftarrow \varphi$ denote the program entry (the `main()` function in C terms), and which correspond to the procedures (recall Figs. 3 and 4). So far, the only solution we see is that this information needs to be communicated to our analyser as part of the input. *Finally*, we observe that so far we evaluate our approach using CHCs that result from reachability questions in relatively simple C programs. These CHCs are also relatively simple and in particular contain at most two predicate applications in the bodies. We wish to evaluate our approach using more complicated CHCs, e.g., that result from cell morphing abstraction [36], but successfully analysing such systems requires to be able to produce disjunctive models.

A Proofs

Proposition 2. *For every $k \geq 1$, the set $\gamma(d_k) \cup \bigcup_{i=1}^{k-1} (\gamma(d_i) \setminus \gamma(b_i))$ is a model of \mathbb{H} .*

Proof. For convenience, let us replace the direct consequence relation $\mathbb{T}_{\mathbb{H}}$ with two objects: the set of initial atoms $\mathbb{I}_{\mathbb{H}} = \{a' \mid (\emptyset, a') \in \mathbb{T}_{\mathbb{H}}\}$ and the set of consecutions $\mathbb{T}_{\mathbb{H}}^{\rightarrow} = \{(A, a') \in \mathbb{T}_{\mathbb{H}} \mid A \neq \emptyset\}$. Then, for every $R, X \subseteq \mathbb{A}$, $\text{post}(\mathbb{T}_{\mathbb{H}}, X) = \mathbb{I}_{\mathbb{H}} \cup \text{post}(\mathbb{T}_{\mathbb{H}}^{\rightarrow}, X)$ and $\text{pre}|_R(\mathbb{T}_{\mathbb{H}}, X) = \text{pre}|_R(\mathbb{T}_{\mathbb{H}}^{\rightarrow}, X)$.

Now let us consider the first three elements of the descending sequence, d_1 , b_1 , and d_2 . For d_1 it holds that $\mathbb{I}_{\mathbb{H}} \cup \text{post}(\mathbb{T}_{\mathbb{H}}^{\rightarrow}, \gamma(d_1)) \subseteq \gamma(d_1)$. That is, $\gamma(d_1)$ is a model of \mathbb{H} and the lemma statement holds for $k = 1$.

For b_1 , it holds that $(\gamma(g) \cap \gamma(d_1)) \cup \text{pre}|_{\gamma(d_1)}(\mathbb{T}_{\mathbb{H}}^{\rightarrow}, \gamma(b_1)) \subseteq \gamma(b_1)$. This means that for every consecution $(A, a') \in \mathbb{T}_{\mathbb{H}}^{\rightarrow}$, if $A \subseteq \gamma(d_1)$ and $A \cap (\gamma(d_1) \setminus \gamma(b_1)) \neq \emptyset$, then $a' \in (\gamma(d_1) \setminus \gamma(b_1))$.

Finally, for d_2 it holds that $(\mathbb{I}_{\mathbb{H}} \cup \text{post}(\mathbb{T}_{\mathbb{H}}^{\rightarrow}, \gamma(d_2))) \cap \gamma(b_1) \subseteq d_2$. First, this means that $\mathbb{I}_{\mathbb{H}} \subseteq (\gamma(d_1) \setminus \gamma(b_1)) \cup \gamma(d_2)$. Indeed, by definition of d_1 , $\mathbb{I}_{\mathbb{H}} \subseteq \gamma(d_1)$ and by definition of d_2 , $\mathbb{I}_{\mathbb{H}} \cap \gamma(b_1) \subseteq \gamma(d_2)$. Second, this means that $\text{post}(\mathbb{T}_{\mathbb{H}}^{\rightarrow}, (\gamma(d_1) \setminus \gamma(b_1)) \cup \gamma(d_2)) \subseteq (\gamma(d_1) \setminus \gamma(b_1)) \cup \gamma(d_2)$. Indeed, let us pick an arbitrary $(A, a') \in \mathbb{T}_{\mathbb{H}}^{\rightarrow}$, s.t. $A \subseteq (\gamma(d_1) \setminus \gamma(b_1)) \cup \gamma(d_2)$. There are two possible cases. If $A \subseteq \gamma(d_2)$ then by definition of d_2 , either $a' \in \gamma(d_2)$, or $a' \in (\gamma(d_1) \setminus \gamma(b_1))$. If $A \not\subseteq \gamma(d_2)$ then $A \cap (\gamma(d_1) \setminus \gamma(b_1)) \neq \emptyset$, and $a' \in \gamma(d_1) \setminus \gamma(b_1)$. This proves the statement of the lemma for $k = 2$ and also provides the base case for the following inductive proof.

Now let $k > 2$, $L_k = \bigcup_{i=1}^{k-1} (\gamma(d_i) \setminus \gamma(b_i))$, and $M_k = \gamma(d_k) \cup L_k$. Let the induction hypothesis be that: $\mathbb{I}_{\mathbb{H}} \subseteq M_k$, $\text{post}(\mathbb{T}_{\mathbb{H}}^{\rightarrow}, M_k) \subseteq M_k$ (i.e., M_k is a model of \mathbb{H}), and for every $(A, a') \in \mathbb{T}_{\mathbb{H}}^{\rightarrow}$, if $A \subseteq M_k$ and $A \cap L_k \neq \emptyset$, then $a' \in L_k$.

Then, let us consider the two subsequent elements: b_k and d_{k+1} and the two sets: $L_{k+1} = M_k \setminus \gamma(b_k)$ and $M_{k+1} = L_{k+1} \cup \gamma(d_{k+1})$.

For b_k it holds that $(\gamma(g) \cap \gamma(d_k)) \cup \text{pre}|_{\gamma(d_k)}(\mathbb{T}_{\mathbb{H}}^{\rightarrow}, \gamma(b_k)) \subseteq \gamma(b_k)$. That is, for every $(A, a') \in \mathbb{T}_{\mathbb{H}}^{\rightarrow}$, if $A \subseteq \gamma(d_k)$ and $A \cap (\gamma(d_k) \setminus \gamma(b_k)) \neq \emptyset$, then $a' \in (\gamma(d_k) \setminus \gamma(b_k))$.

For d_{k+1} it holds that $(\mathbb{I}_{\mathbb{H}} \cup \text{post}(\mathbb{T}_{\mathbb{H}}^{\rightarrow}, \gamma(d_{k+1}))) \cap \gamma(b_k) \subseteq \gamma(d_{k+1})$.

First, observe that $\mathbb{I}_{\mathbb{H}} \subseteq M_{k+1}$. Indeed, we know that $\mathbb{I}_{\mathbb{H}} \subseteq M_k$ and that $M_{k+1} = (M_k \setminus \gamma(b_k)) \cup \gamma(d_{k+1})$. By definition of d_{k+1} , $\mathbb{I}_{\mathbb{H}} \cap \gamma(b_k) \subseteq \gamma(d_{k+1})$. Thus, $\mathbb{I}_{\mathbb{H}} \subseteq M_{k+1}$.

Second, let us pick an arbitrary $(A, a') \in \mathbb{T}_{\mathbb{H}}^{\rightarrow}$, s.t. $A \subseteq M_{k+1}$. Since M_k is a model of H , we know that $a' \in M_k$. But then, there are three possible cases. (i) If $A \subseteq \gamma(d_{k+1})$, then either $a' \in \gamma(d_{k+1})$, or $a' \notin \gamma(b_k)$. That is, $a' \in (M_k \setminus \gamma(b_k)) \cup \gamma(d_{k+1}) = M_{k+1}$. (ii) If $A \subseteq \gamma(d_k)$ and $A \not\subseteq \gamma(d_{k+1})$, then $A \cap (\gamma(d_k) \setminus \gamma(b_k)) \neq \emptyset$, and $a' \in \gamma(d_k) \setminus \gamma(b_k) \subseteq M_{k+1}$. (iii) Finally, if $A \not\subseteq \gamma(d_k)$, then $A \cap L_k \neq \emptyset$, and from the hypothesis $a' \in L_k$. There are no other possible cases. This means that $\text{post}(\mathbb{T}_{\mathbb{H}}^{\rightarrow}, M_{k+1}) \subseteq M_{k+1}$ and thus M_{k+1} is a model of \mathbb{H} . Also, from (ii) and (iii) it follows that for $(A, a') \in \mathbb{T}_{\mathbb{H}}^{\rightarrow}$, if $A \subseteq M_{k+1}$ and $A \cap L_{k+1} \neq \emptyset$, then $a' \in L_{k+1}$.

References

1. Clang: a C language family frontend for LLVM. <https://clang.llvm.org/>. Accessed July 2017
2. Competition on software verification (SV-COMP). <http://sv-comp.sosy-lab.org/>. Accessed July 2017
3. Frama-C software analyzers. <https://frama-c.com/>. Accessed July 2017
4. A path focusing abstract interpreter for horn clauses. <https://gitlab.com/abakhirkin/hcai>. Accessed July 2017
5. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice. Programming and Software Engineering, vol. 10001. Springer, Heidelberg (2016). doi:10.1007/978-3-319-49812-6
6. Apinis, K., Seidl, H., Vojdani, V.: Side-effecting constraint systems: a swiss army knife for program analysis. In: Jhala, R., Igarashi, A. (eds.) APLAS 2012. LNCS, vol. 7705, pp. 157–172. Springer, Heidelberg (2012). doi:10.1007/978-3-642-35182-2_12
7. Bagnara, R., Hill, P.M., Zaffanella, E.: The parma polyhedra library: toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Sci. Comput. Program. **72**(1–2), 3–21 (2008)
8. Benoy, F., King, A.: Inferring argument size relationships with CLP(\mathcal{R}). In: Gallagher, J. (ed.) LOPSTR 1996. LNCS, vol. 1207, pp. 204–223. Springer, Heidelberg (1997). doi:10.1007/3-540-62718-9_12
9. Beyene, T.A., Popeea, C., Rybalchenko, A.: Solving existentially quantified horn clauses. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 869–882. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39799-8_61
10. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, Austin, Texas, USA, pp. 25–32. IEEE, 15–18 November 2009
11. Bjørner, N., Gurfinkel, A., McMillan, K., Rybalchenko, A.: Horn clause solvers for program verification. In: Beklemishev, L.D., Blass, A., Dershowitz, N., Finkbeiner, B., Schulte, W. (eds.) Fields of Logic and Computation II. LNCS, vol. 9300, pp. 24–51. Springer, Cham (2015). doi:10.1007/978-3-319-23534-9_2

12. Bjørner, N., de Moura, L., Wintersteiger, C.: Z3. <https://github.com/Z3Prover/z3>. Accessed July 2017
13. Bourdoncle, F.: Sémantiques des langages impératifs d'ordre supérieur et interprétation abstraite. Ph.D. thesis, École polytechnique (1992)
14. Bourdoncle, F.: Efficient chaotic iteration strategies with widenings. In: Bjørner, D., Broy, M., Pottosin, I.V. (eds.) *Formal Methods in Programming and Their Applications*. LNCS, pp. 128–141. Springer, Heidelberg (1993). doi:[10.1007/BFb0039704](https://doi.org/10.1007/BFb0039704)
15. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R.M., Harrison, M.A., Sethi, R. (eds.) *Principles of Programming Languages (POPL)*, pp. 238–252. ACM (1977)
16. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Aho, A.V., Zilles, S.N., Rosen, B.K. (eds.) *Principles of Programming Languages (POPL)*, pp. 269–282. ACM Press (1979)
17. Cousot, P., Cousot, R.: Abstract interpretation and application to logic programs. *J. Log. Program.* **13**(2–3), 103–179 (1992)
18. Cousot, P., Cousot, R.: Refining model checking by abstract interpretation. *Autom. Softw. Eng.* **6**(1), 69–95 (1999)
19. De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: Program verification via iterated specialization. *Sci. Comput. Program.* **95**, 149–175 (2014)
20. Gawlitza, T.M., Seidl, H.: Precise program analysis through strategy iteration and optimization. In: Nipkow, T., Grumberg, O., Hauptmann, B. (eds.) *Software Safety and Security - Tools for Analysis and Verification*, NATO Science for Peace and Security Series - D: Information and Communication Security, vol. 33, pp. 348–384. IOS Press (2012)
21. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: Vitek, J., Lin, H., Tip, F. (eds.) *Programming Language Design and Implementation (PLDI)*, pp. 405–416. ACM (2012)
22. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The seahorn verification framework. In: Kroening, D., Păsăreanu, C.S. (eds.) *CAV 2015*. LNCS, vol. 9206, pp. 343–361. Springer, Cham (2015). doi:[10.1007/978-3-319-21690-4_20](https://doi.org/10.1007/978-3-319-21690-4_20)
23. Henry, J., Monniaux, D., Moy, M.: PAGAI: a path sensitive static analyser. *Electron. Notes Theor. Comput. Sci.* **289**, 15–25 (2012)
24. Henry, J., Monniaux, D., Moy, M.: Succinct representations for abstract interpretation. In: Miné, A., Schmidt, D. (eds.) *SAS 2012*. LNCS, vol. 7460, pp. 283–299. Springer, Heidelberg (2012). doi:[10.1007/978-3-642-33125-1_20](https://doi.org/10.1007/978-3-642-33125-1_20)
25. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Cimatti, A., Sebastiani, R. (eds.) *SAT 2012*. LNCS, vol. 7317, pp. 157–171. Springer, Heidelberg (2012). doi:[10.1007/978-3-642-31612-8_13](https://doi.org/10.1007/978-3-642-31612-8_13)
26. Jaffar, J., Maher, M.J.: Constraint logic programming: a survey. *J. Log. Program.* **19**(20), 503–581 (1994)
27. Jeannet, B.: Bddapron. <http://pop-art.inrialpes.fr/~bjeannet/bjeannet-forge/bddapron/>. Accessed July 2017
28. Kafle, B., Gallagher, J.P.: Constraint specialisation in horn clause verification. In: Asai, K., Sagonas, K. (eds.) *Partial Evaluation and Program Manipulation (PEPM)*, pp. 85–90. ACM (2015)
29. Kafle, B., Gallagher, J.P.: Tree automata-based refinement with application to horn clause verification. In: D'Souza, D., Lal, A., Larsen, K.G. (eds.) *VMCAI 2015*. LNCS, vol. 8931, pp. 209–226. Springer, Heidelberg (2015). doi:[10.1007/978-3-662-46081-8_12](https://doi.org/10.1007/978-3-662-46081-8_12)

30. Kafle, B., Gallagher, J.P., Morales, J.F.: RAHFT: a tool for verifying horn clauses using abstract interpretation and finite tree automata. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 261–268. Springer, Cham (2016). doi:[10.1007/978-3-319-41528-4_14](https://doi.org/10.1007/978-3-319-41528-4_14)
31. Karpenkov, E.G., Monniaux, D., Wendler, P.: Program analysis with local policy iteration. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 127–146. Springer, Heidelberg (2016). doi:[10.1007/978-3-662-49122-5_6](https://doi.org/10.1007/978-3-662-49122-5_6)
32. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 17–34. Springer, Cham (2014). doi:[10.1007/978-3-319-08867-9_2](https://doi.org/10.1007/978-3-319-08867-9_2)
33. Komuravelli, A., Gurfinkel, A., Chaki, S., Clarke, E.M.: Automatic abstraction in SMT-based unbounded software model checking. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 846–862. Springer, Heidelberg (2013). doi:[10.1007/978-3-642-39799-8_59](https://doi.org/10.1007/978-3-642-39799-8_59)
34. Marques-Silva, J., Janota, M., Belov, A.: Minimal sets over monotone predicates in boolean formulae. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 592–607. Springer, Heidelberg (2013). doi:[10.1007/978-3-642-39799-8_39](https://doi.org/10.1007/978-3-642-39799-8_39)
35. Monniaux, D., Gonnord, L.: Using bounded model checking to focus fixpoint iterations. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 369–385. Springer, Heidelberg (2011). doi:[10.1007/978-3-642-23702-7_27](https://doi.org/10.1007/978-3-642-23702-7_27)
36. Monniaux, D., Gonnord, L.: Cell morphing: from array programs to array-free horn clauses. In: Rival, X. (ed.) SAS 2016. LNCS, vol. 9837, pp. 361–382. Springer, Heidelberg (2016). doi:[10.1007/978-3-662-53413-7_18](https://doi.org/10.1007/978-3-662-53413-7_18)
37. Peralta, J.C., Gallagher, J.P.: Convex hull abstractions in specialization of CLP programs. In: Leuschel, M. (ed.) LOPSTR 2002. LNCS, vol. 2664, pp. 90–108. Springer, Heidelberg (2003). doi:[10.1007/3-540-45013-0_8](https://doi.org/10.1007/3-540-45013-0_8)
38. Rival, X., Mauborgne, L.: The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.* **29**(5), 26 (2007)
39. Rümmer, P., Hojjat, H., Kuncak, V.: Classifying and solving horn clauses for verification. In: Cohen, E., Rybalchenko, A. (eds.) VSTTE 2013. LNCS, vol. 8164, pp. 1–21. Springer, Heidelberg (2014). doi:[10.1007/978-3-642-54108-7_1](https://doi.org/10.1007/978-3-642-54108-7_1)
40. Sharygina, N., Veith, H. (eds.): CAV 2013. LNCS, vol. 8044. Springer, Heidelberg (2013)



<http://www.springer.com/978-3-319-66705-8>

Static Analysis

24th International Symposium, SAS 2017, New York, NY,
USA, August 30 - September 1, 2017, Proceedings

Ranzato, F. (Ed.)

2017, XIV, 473 p. 114 illus., Softcover

ISBN: 978-3-319-66705-8