Non-Monotonic Program Analysis

Daniel Schwartz-Narbonne
New York University

Philipp Rümmer
Uppsala University

Martin Schäf
SRI International

Ashish Tiwari
SRI International

Thomas Wies
New York University

Traditionally, program analysis is formulated as computation of fixpoints using monotonic iteration of lattice-theoretic functions. Monotonicity is important because it ensures convergence of the analysis towards a fixpoint. Still, the idea of non-monotonic iteration is intriguing because such an analysis can cut short the search, potentially converging much faster than monotonic iteration. In this paper, we answer the question whether non-monotonic analyses are a worthwhile pursuit. We consider several non-monotonic algorithms for the specific problem of solving systems of Horn clauses. Our algorithms have in common that they (1) use logical abduction to span the search space of non-monotonic iteration sequences, and (2) bound the non-monotonic search by a monotone sequence of checkpoints to enforce overall convergence. The algorithms differ in their search strategies, where the most interesting one performs an A*-like search. We have implemented these algorithms and compared them against existing monotonic analyses for solving Horn clauses. Our evaluation indicates that non-monotonic fixpoint iteration is a promising complementary technique to traditional program analyses.

1 Introduction

Program analysis is quintessentially a fixpoint computation problem [8]. Constructive definitions of fixpoints are given in terms of a monotonic sequence whose limit is the required fixpoint. Most program analysis techniques compute such a monotonic sequence to find fixpoints. We call them monotonic program analyses here. Can we build program analyzers that are non-monotonic? In other words, in our effort to find a desired fixpoint, can we generate a non-monotonic sequence that eventually leads to the fixpoint? Could the non-monotonic search expedite our convergence to the fixpoint? We try to answer these questions in this paper.

In this paper, we are interested in the assertion checking problem: given a program and an assertion, prove that the assertion is valid on all executions of the program. We can solve the assertion checking problem by finding an invariant that is sufficient to prove the assertion. An invariant is a set of states that is a fixpoint (of the one-step transition relation describing the, possibly abstract, semantics) of the program. There are two different ways to find a fixpoint (invariant):

Increasing Sequence. We start with an underapproximation S_0 of the invariant, iteratively *add* states to the set to generate an increasing chain of sets

$$S_0 \subset S_1 \subset S_2 \subset \ldots$$

until we reach a fixpoint.

Decreasing Sequence. We start with an overapproximation T_0 of the desired invariant, iteratively *remove* states from the set to generate a decreasing chain of sets

$$T_0 \supset T_1 \supset T_2 \supset \ldots$$

Submitted to:
HCVS 2015

until we reach a fixpoint.

Typically, the set S_0 consists of all possible initial states of the program in the *increasing sequence* approach, and the sets $S_1, S_2, \ldots$ are generated via *forward propagation* using (approximations of) the strongest postcondition operator. On the other hand, in the *decreasing sequence* approach, the set T_0 consists of all the (assumed) safe states specified in the assertion, and the sets $T_1, T_2, \ldots$ are generated via *backward propagation* using (approximations of) the weakest precondition operator. Hence, we shall refer to the increasing sequence approach as forward analysis, and the decreasing sequence approach as backward analysis.

Both forward and backward analysis are monotonic, as they generate intermediate sets that are monotonically increasing or decreasing. Both have their own benefits. Forward analysis is guided by the initial program states, but remains oblivious to the (final) assertion all through. On the other hand, backward analysis is goal-directed, but it is unable to benefit from the knowledge of the initial states in its entire run. Hence, both are not perfect in the search for the desired fixpoint (invariant). By combining forward and backward analysis, one could possibly obtain a program analysis technique that is goal-directed, but also cognizant of the initialization. Hence, a combined approach can possibly reach fixpoint faster than either individual approach. However, a combined approach will generate a non-monotonic sequence of sets, which can possibly cycle without making progress toward reaching a fixpoint.

In this paper, we investigate non-monotonic approaches to program analysis that combine both forward and backward analyses. We use the Horn clause formulation of program analysis problems to present our approach. However, our ideas are more generally applicable and can be lifted to program analysis that are described using other means. Our main contribution is that we present different non-monotonic approaches that combine forward and backward analysis, which are all guaranteed to make progress. One of the main conclusion of our study is that non-monotonic analysis can often reach fixpoints faster than their monotonic counterparts. Moreover, it is possible to use several different strategies for combining forward and backward steps, where some of the interesting and effective strategies perform A*-like search.

1.1 Brief Overview of our Approach

The search for a suitable invariant that is sufficient to verify assertions in a program can be formulated as a search for a satisfying assignment of a set of Horn clauses that contain relational variables (higher-order variables that represent the "invariant" at different program points). The Horn clause formulation is appealing since it can uniformly describe analysis of iterative and recursive programs. It also allows flexibility in reasoning about different paths in the program. Furthermore, analysis approaches map directly to solving of Horn clauses: forward propagation involves updating the (relation in the) head of the Horn clause, whereas backward propagation involves updating the (relations in the) body of the Horn clause.

We illustrate our main idea behind ensuring progress of our non-monotonic procedure in Figure 1. The notional plot in Figure 1 shows that in non-monotonic program analysis, we allow the procedure to search the fixpoint by using an arbitrary strategy to alternate between forward and backward steps. To avoid getting stuck in a cycle, we use the following key idea: every so often, we mark partial solutions that cannot be improved by a backward (strengthening) step alone, and force these marked solutions to monotonically get bigger (by disjuncting them with the previously marked solution). Thus, we get a monotonically increasing set – shown by the bottom dotted line in Figure 1. We can also perform the dual operation: every so often, we mark partial solutions that cannot be improved by a forward (weakening) step alone, and force these marked solutions to monotonically get smaller (by conjuncting them with the

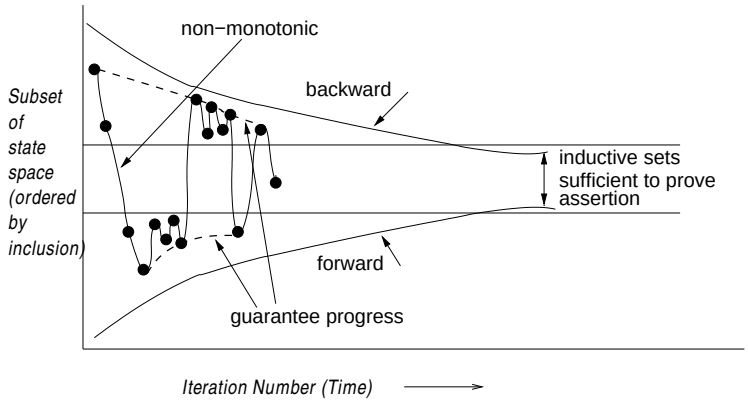

Figure 1: Notional illustration of non-monotonic analysis: Unlike forward or backward analysis that monotonically advance toward a fixpoint, non-monotonic analysis can fluctuate in arbitrary ways. Progress is guaranteed by bounding the fluctuations inside a monotonically shrinking region (shown by dotted lines).

previously marked solution). Thus, we get a monotonically decreasing set – shown by the top dotted line in Figure 1. Doing either one is sufficient to guarantee progress. In between these marked solution points, the algorithm is free to pick any strategy to alternate between forward and backward steps. In particular, a A*-heuristic search can be implemented that can potentially guide convergence to the fixpoint much faster.

2 Related Work

There is plenty of work in combining forward and backward analyses; for instance, see [6, 9, 24]. In all of this existing work, forward and backward analyses are combined in stages in a careful way – typically, each monotonic analysis is performed "to completion" before switching to the dual analysis (which is used to refine the results from the other analyses). One truly non-monotonic program analysis approach was proposed by Gulwani and Jojic [16], where the hypothesized invariants at each program point were made "less locally inconsistent" in every step, without any regards to preserving monotonicity, and in a probabilistic fashion guided by an "inconsistency measure". Our procedure is similar and inherits many of the benefits of that approach [16]. One difference is that our formulation uses Horn clauses and abduction, and hence, our analysis can be more fine-grained and focus on individual paths. Second, in each step, we make hypothesized invariants "locally consistent", whereas [16] can leave them inconsistent. Moreover, [16] can be viewed as performing probabilistic inference (Gibbs sampling) guided by the inconsistency measure, whereas our algorithm is deterministic (but it can use A* search to guide the process of convergence).

Logic abduction, originally introduced by [29], has recently emerged as a new technique to derive annotations for program analysis. Algorithms have been presented that use abduction to improve shape analysis [7], for under-approximation in abstract interpretation [17], and for fault-localization based on abduction [12]. More recently, abduction has been used to automatically infer loop invariants [13]. The algorithm in this paper generalizes the idea of using abduction to find loop invariants to solving systems of Horn clauses, for instance for the purpose of analysing recursive programs.

Horn clauses were proposed as intermediate representation for verification in a number of recent papers, including [4,15,27]. [18] uses Horn clauses for verification of multi-threaded programs. A range of applications of Horn clauses, including inter-procedural model checking, was given in [15]. Extensions of Horn clauses used in verification include variants with universal [5] and existential quantifiers [3], as well as alternations [2].

One benefit of formulating program analysis problems in terms of logic programming is that existing analysis techniques based on constraint solving and theorem proving can be rephrased for Horn clauses, yielding useful generalizations. For example, rephrasing interpolation-based loop invariant inference [14, 25] for Horn clauses leads to the more general notion of tree interpolants [32]. Tree interpolation is no longer restricted to the inference of invariants for sequential loops but also applies to inference problems in recursive and multi-threaded programs.

In the context of verification, the majority of algorithms to solve Horn clauses are based on *Craig interpolation*, and use solvers for recursion-free Horn clauses as a sub-procedure. The procedure for solving Horn clauses from [18], over the combined theory of linear integer arithmetic and uninterpreted functions, was presented in [19]. Further algorithms were developed in [26, 32], along with a range of generalised methods for Craig interpolation, such as tree interpolants, DAG interpolants, and disjunctive interpolants. A further line of research applies *incremental induction* (IC3/PDR) to solve Horn clauses [23]: fixed-point constraints are solved by incrementally strengthening approximations of the sets of states reachable in $i = 1,\ldots,k$ steps, until eventually a sufficiently strong inductive invariant is found.

Inter-procedural software model checking with function summaries and interpolation has been an active area of research over the last decade; we briefly survey techniques that are most related to our work. In the context of predicate abstraction, it has been discussed how well-scoped invariants can be inferred [22] in the presence of function calls. Based on the concept of Horn clauses, a predicate abstraction-based algorithm for bottom-up construction of function summaries was presented in [15]. Function summaries generated using interpolants have also been used to speed up bounded model checking [34]. Generalisations of the Impact algorithm [25] to programs with procedures are given in [21] (formulated using nested word automata) and [1].

Synthesis of necessary pre-conditions of a program has been described, among others, in [10], [28], and [30].

3 Preliminaries

Throughout this paper, we assume that a first-order vocabulary of *interpreted symbols* has been fixed, consisting of a set $\mathscr{F}$ of fixed-arity function symbols, and a set $\mathscr{P}$ of fixed-arity predicate symbols. Interpretation of $\mathscr{F}$ and $\mathscr{P}$ is determined by a class $\mathscr{S}$ of structures (U,I) consisting of non-empty universe U, and a mapping I that assigns to each function in $\mathscr{F}$ a set-theoretic function over U, and to each predicate in $\mathscr{P}$ a set-theoretic relation over U. As a convention, we assume the presence of an equation symbol "=" in $\mathscr{P}$, with the usual interpretation. Given a countably infinite set $\mathscr{X}$ of variables, a *constraint language* is a set *Constr* of first-order formulae over $\mathscr{F}, \mathscr{P}, \mathscr{X}$ For example, the language of quantifier-free Presburger arithmetic has $\mathscr{F} = \{+,-,0,1,2,\ldots\}$ and $\mathscr{P} = \{=,\leq,|\}$).

A constraint is called *satisfiable* if it holds for some structure in $\mathscr{S}$ and some assignment of the variables $\mathscr{X}$, otherwise *unsatisfiable*. We say that a set $\Gamma \subseteq$ *Constr* of constraints *entails* a constraint $\phi \in$

Constr if every structure and variable assignment that satisfies all constraints in Γ also satisfies ϕ; this is denoted by $\Gamma \models \phi$. When Γ is empty, we just write $\models \phi$ and say that ϕ is *valid*.

Given a constraint ϕ, $fv(\phi)$ denotes the set of free variables in ϕ. We write $\phi[x_1,\ldots,x_n]$ to state that a constraint contains (only) the free variables $x_1,\ldots,x_n$, and $\phi[t_1,\ldots,t_n]$ for the result of substituting the terms $t_1,\ldots,t_n$ for $x_1,\ldots,x_n$.

Given a constraint ϕ, and a set $V \subseteq \mathscr{X}$ of variables, we define the formula $\exists_{fv(\phi)\setminus V}.\ \phi$ to be the *existential projection* $proj_{\exists}(\phi,V)$ of ϕ onto V.

Abduction is a technique to infer a missing premise to explain a given conclusion. Formally, given an axiom A and a conclusion B, an abduction for A and B is a constraint ϕ such that $A \wedge \phi \models B$, and $A \wedge \phi$ is satisfiable. Throughout the paper we use an abductive inference algorithm based on the one by Dillig et al [13]: the problem of finding an abduction ϕ, such that $\models (A \wedge \phi) \implies B$ can be rewritten as $\phi \models (A \implies B)$. Now, it is evident that we can find an abduction by universally quantifying any subset $V \subseteq fv((A \implies B))$ of free variables in $(A \implies B)$. That is, since, for any such V, we have

$$(\forall V.A \implies B) \models (A \implies B)$$

any formula ϕ which is logically equivalent to $\forall V.A \implies B$ is a valid abduction as long as it does not contradict with A.

In [13], sets V of maximum cardinality (such that $\forall V.A \implies B$ and A are consistent) are selected to compute abductions. For our analysis approach, we found it beneficial to consider a larger range of abductions by selecting *all maximum* sets V such that $\forall V.A \implies B$ and A are consistent; the overhead of computing more abductions was easily compensated by the greater flexibility to generalize and find better predicates.

3.1 Horn Clauses

To define Horn clauses that can encode program verification problems, we fix a set $\mathscr{R}$ of uninterpreted fixed-arity *relation symbols,* disjoint from $\mathscr{P}$ and $\mathscr{F}$. A *Horn clause* is a formula $C \wedge B_1 \wedge \cdots \wedge B_n \to H$ where

- C is a constraint over $\mathscr{F}, \mathscr{P}, \mathscr{X}$;
- each B_i is an application $p(t_1,\ldots,t_k)$ of a relation symbol $p \in \mathscr{R}$ to first-order terms over $\mathscr{F}, \mathscr{X}$;
- H is similarly either an application $p(t_1,\ldots,t_k)$ of $p \in \mathscr{R}$ to first-order terms, or is the constraint *false*.

H is called the *head* of the clause, $C \wedge B_1 \wedge \cdots \wedge B_n$ the *body.* In case $C = true$, we usually leave out C and just write $B_1 \wedge \cdots \wedge B_n \to H$. First-order variables (from $\mathscr{X}$) in a clause are considered implicitly universally quantified; relation symbols represent set-theoretic relations over the universe U of a structure $(U,I) \in \mathscr{S}$. Notions like (un)satisfiability and entailment generalise straightforwardly to formulae with relation symbols.

A *relation symbol mapping* is a function $m : \mathscr{R} \to Constr$ that maps each n-ary relation symbol $p \in \mathscr{R}$ to a constraint $M(p) = C_p[x_1,\ldots,x_n]$ with n free variables. The *instantiation* $m(h)$ of a Horn clause h is defined by:

$$m\big(C \wedge p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n) \to p(\bar{t})\big) = C \wedge m(p_1)[\bar{t}_1] \wedge \cdots \wedge m(p_n)[\bar{t}_n] \to m(p)[\bar{t}]$$
$$m\big(C \wedge p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n) \to false\big) = C \wedge m(p_1)[\bar{t}_1] \wedge \cdots \wedge m(p_n)[\bar{t}_n] \to false$$

Let $\mathscr{HC}$ be a set of Horn clauses over relation symbols $\mathscr{R}$. A $\mathscr{HC}$ is called *solvable* if there is a relation symbol mapping m such that the instantiation $m(h)$ of every clause $h \in \mathscr{HC}$ is valid.

Following recent work [4], it is clear that the assertion checking problem can be reduced to solvability of Horn clauses.

4 Solving Horn Clauses with Abduction

In this section, we present our approach for assertion checking based on solving Horn clauses.

We assume we are given a set of Horn clauses $\mathscr{HC}$, where each clause $h \in \mathscr{HC}$ is satisfiable, but not necessarily valid. We want to find a relation symbol mapping m that solves the Horn clauses. We find such an m by starting with an initial (guess) mapping m_0 and updating it in every step until we find the desired m. We use abduction to perform the update.

Let m be the current guess. If $m(h)$ is valid for every $h \in \mathscr{HC}$, we are done. Therefore, let h be a Horn clause such that $m(h)$ is not valid. Given the Horn clause h and the guess m, the *abduction procedure* returns a set of formulas $abd_0, abd_1 \ldots$ such that for each abd_j in this set, $m(abd_j \wedge h)$ is valid. Let h be of the form $C \wedge B_1 \wedge \cdots \wedge B_n \rightarrow H$. We can now use the formula abd (from the set $\{abd_0, abd_1, \ldots\}$) to update the predicates in h to get the next guess m'. To that end, we have two options: we can weaken the predicate in the head of the clause, or we can strengthen the predicates in the body of the clause.

Weakening. Weakening is the simpler option. Since, by definition, there is only one predicate H in the head of the Horn clause, we can update the current mapping of H to:

$$m'(H) \mapsto (abd \implies m(H)).$$

Strengthening. The second option is to strengthen the predicates in $B_1 \wedge \cdots \wedge B_n$ to attempt to ensure that

$$\models m'(B_1 \wedge \cdots \wedge B_n) \rightarrow abd.$$

There are many possible ways of partitioning the abduction across the B's. In our implementation, we strengthen each B_i by conjoining it with the projection of abd onto the variables of B_i, where projection is defined as existentially quantifying out any variables that appear in abd but not in B_i. That is, for each B_i, we update our mapping as follows:

$$m'(B_i) \mapsto (proj(abd, vars(B_i)) \wedge m(B_i)).$$

Both, weakening and strengthening can be used to modify the assignments of the predicates in $\mathscr{HC}$ such that a previously non-valid clause become valid. However, such an update might turn a previously valid clause into a non-valid one. That is, making all clauses in $\mathscr{HC}$ valid can be seen as a search problem where we try to find (possibly short) sequence of weakening- and/or strengthening-steps that make all clauses valid.

In the following, we discuss several search strategies to solve this problem, discuss their benefits and problems and compare them experimentally. In the following we do not make any assumption about the abduction procedure other than that it always finds at least one abduction if one exists.

4.1 Breadth-first Strategy

The Breadth-first strategy takes a very simple solution to this problem: try every candidate solution in order.

We can consider the candidate assignments (m' above) generated by the abduction procedure as forming a tree, where a node represents a candidate assignment. Leaf nodes are either true solutions to the horn clause problem, or assignments for which the abduction procedure fails to generate new assignments. For each assignment, the abduction procedure generates a finite number of candidate weakenings and strengthenings, so each node has a finite number of children, and hence at every level, the tree has finite width. The depth of the tree, however, may be unbounded (for example, the procedure might guess the relation $x > 4$, then $x > 5$, then $x > 6$ etc).

The breadth-first strategy uses a FIFO queue to visit the tree in breadth first order. A candidate assignment is removed from the queue and checked for correctness. If it is correct, we return success; if not, we use abduction to generate its children, and place them on the queue. The key advantage behind the breadth first strategy is that it guarantees a form of completeness: if a solution can be deduced using the abduction procedure, it will be reached in a finite (albeit potentially exponentially large) number of steps. Note that this is *not* a complete verification procedure. The abduction procedure may fail to produce a necessary assignment, in which case we are incomplete.

We implemented two optimizations to the standard BFS algorithm. Since checking the correctness of an candidate assignment is relatively cheap, we check candidate assignments as soon as they are generated, and return success if we find one that satisfies the problem.

The second optimization came from the observation that candidate assignments tended to reappear in the search. The result of strengthening and then weakening an assignment is sometimes (although not always) the original assignment again. We maintain a set of previously processed candidate assignments, and only process new assignments that are not members of this set.

4.2 Depth-first Strategy

Like for the breadth-first strategy, we think of the candidate assignments generated by the abduction procedure as forming a tree, where a node represents a candidate assignment. The depth-first strategy explores the tree until it reaches a leaf node. We then check if this node is indeed a solution to the horn clause problem, and if not, we backtrack.

Since the depth of the tree of possible abductions is unbounded, this strategy does not carry the same completeness guarantee as the breadth-first strategy. However, it may avoid visiting candidate assignments that do not lead to the solution.

We prune the size of the search tree by maintaining an upper bound for each search path (upper dotted line in Figure 1). This allows us to eliminate certain weakening steps: if a weakening step would exceed the upper bound, we return the upper bound instead, and recurse on another path.

Such an upper bound, initially being true for each predicate assignment, can be updated every time we do not find an abduction to further weaken an assignment. In that case, we update the upper bound to this assignment intersected with the previous upper bound. By always intersecting the new upper bound with the previous one, we ensure monotonicity and therefore progress (see Figure 1).

4.3 A^* Strategy

The A^* strategy is inspired by the A^* graph searching algorithm [20]. A^* modifies BFS by replacing the FIFO queue with a priority queue, where the priorities are determined heuristically. This ensures that

nodes that, according to the heuristic, are likely to lead to the answer are visited first. If the heuristic is well chosen, this can dramatically improve the speed with which the solution is found.[1]

4.3.1 Selecting the Priority Heuristic

We considered two factors in developing our heuristic. First of all, we should prefer candidate solutions which already solve more clauses: all else being equal, we are much more likely to reach the solution from a candidate in which only one clause is unsatisfied than one in which half of the clauses are unsatisfied.

Secondly, our experience shows that successful candidates assignments tend to be syntactically simple. Candidates with extremely complicated formulas tend to represent cases where the abduction procedure starts to iterate constants, such as $x \neq 1 \wedge x \neq 2 \wedge \ldots$, which often does not converge in a reasonable time.

We measured the complexity of a formula as the number of operators used (conjuncts, disjuncts, additions, subtraction, etc) to represent the formula. $(a > b)$ would have syntactic complexity of 1, whereas

$$((a > b) \wedge (x + y < 12))$$

would have syntactic complexity of 4.

We tested a number of combinations of these two heuristics, and discovered that a simple multiplicative factor

$$(\text{\#unsolved clauses}) * (\text{syntatic complexity})$$

seemed to be the most effective.

We maintained the existing optimizations from the BFS algorithm (early checking, and a set of previously processed solutions to prevent repeating).

5 Experimental Evaluation

We perform a feasibility study of our approach and compare the different search strategies on a set of common benchmark problems. As a baseline, we use two established tools for solving horn clauses, Z3 [5] and Eldarica [32]. For the experimental evaluation, we implemented the breadth-first strategy from Section 4.1 as *BFS*, the depth-first strategy from Section 4.2 as *DFS*, and the A^* strategy from Section 4.3 as *AST*.[2]

Experimental Setup. For solving horn clauses, we use the Princess theorem prover [31] and an implementation of the abduction procedure described in [11]. The search strategies from the previous section are implemented as small Scala programs on top the abduction procedure and the theorem prover.

We run each search strategy as well as Z3 and Eldarica for 5 minutes on each Benchmark problem. If no solution is found, we mark the attempt as a timeout. Since all approaches are sound, we only compare whether they are able to find a solution and stop the time it takes to get there. As discussed later in the threats to validity, we rerun our experiments with a larger timeout to ensure that this timeout is not biased towards any of the tools.

[1] In the ideal case, if the heuristic is within a bound of the correct cost, (technically, an "admisible heuristic"), A^* search is provably optimal.

[2] Tool and benchmarks are available at `www.csl.sri.com/~schaef/fm2015.tar`

The benchmark set consists of 81 horn clause problems generated from looping or recursive C code followed by an assertion that needs to be proved. The first 47 benchmarks prefixed with "loop" are provided by the authors of [13]. Unfortunately, the tool implemented in this paper was not available for comparison. The next 18 benchmarks prefixed with "RECUR" are taken from the SVCOMP benchmark set.[3] The remaining benchmarks are manual encodings of popular algorithms such as towers of Hanoi, computation of Fibonacci numbers, or if a number is even or odd. Some of these are similar to examples in the loop- or SVCOMP benchmarks, but use a different horn clause encoding.

Research Question. The goal of our experiment is to check if our non-monotonic approach can solve benchmark problems that cannot be solved by Z3 and Eldarica. In general, however, we have to curb our enthusiasm: if either Z3 or Eldarica find a solution then there exists a solution that can be reached with a monotonic analysis within the time limit. Hence our approach can, at best, also find this solution but is unlikely to be faster as our search space is bigger. Hence, the question that we are interested in is if non-monotonic analysis can solve horn clause problems that cannot be solved by monotonic approaches.

Results. Table 1 gives a refined view of the experimental results. for space reasons, we refer to the appendix for a per-benchmark evaluation of the results. From our strategies, the *AST* approach performs best. It solves 37 of 81 benchmark problems. Any benchmark solved by *DFS* could also be solved by *AST*, but *AST* is able to solve 11 benchmarks that *DFS* could not solve. Further, *AST* solves 17 benchmarks that cannot be solved by *BFS*. However, *BFS* is able to solve one benchmark where *AST* times out. This is an example that our A^* heuristic of preferring simpler abductions is not a perfect choice and that there is room for improvement.

Z3 solves 54 benchmark problems, and Eldarica solves 55. Our approach solves two benchmarks that can neither be solved by Z3 nor by Eldarica. Further, 5 of the benchmarks solved by us cannot be solved by Z3 and 8 benchmarks cannot be solved by Eldarica. Z3 solves 13 benchmarks that cannot be solved by Eldarica. Eldarica in turn can solve 14 benchmarks that cannot be handled by Z3. Out of the 81 benchmarks, 11 cannot be solved by any approach.

	Z3	Eldarica	*BFS*	*DFS*	*AST*
solved	54	55	21	26	37
solved exclusively	6	12	2		
Not by Z3	-	14	2	5	5
Not by Eldarica	13	-	8	7	8
Not by *BFS*	35	42	-	10	17
Not by *DFS*	33	36	5	-	11
Not by *AST*	22	26	1	0	-

Table 1: The number of benchmarks solved per tool. This first row shows the total number of benchmarks solved, the second shows the number of benchmarks that only this tool could solve, and the other rows show how many benchmarks one approach could solve that another couldn't solve.

Discussion. Our experiments show that our approach can solve an interesting subset of benchmarks that cannot be solved by Z3 or Eldarica. The experiments further show that using a heuristic yields

[3] `https://svn.sosy-lab.org/software/sv-benchmarks/trunk/clauses/LIA/Eldarica/RECUR/`

substantial improvements in the number of solved benchmarks. The *AST* approach solves significantly more benchmarks than *BFS* and *DFS* while there is only one benchmark where *AST* is worse than *BFS*. Hence, we are optimistic that future improvements to the heuristic will allow us to solve more of the benchmarks.

From in-depth analysis of cases where our approach fails, we identified the abduction procedure as the main bottleneck. In many cases where our approach fails to find a solution it fails because the abduction procedure does not generate the right predicates. With an ideal abduction procedure, these benchmarks would easily be solvable. We believe that further improvements on the abduction procedure will allow us to solve a larger number of these benchmarks; in particular, we plan to develop abduction procedures that are able to take multiple clauses into account, to obtain a more holistic picture of the problem to be solved. Further improvements can be expected by integrating orthogonal existing techniques, including acceleration and widening methods. It is also planned to investigate whether abduction can be combined with the exploration technique from [33].

On the benchmarks that can be solved by us and Z3 or Eldarica, our implementation is usually slower than the other tools. However, there are several simple techniques that will yield substantial speed-ups. For example, our current implementation checks if it already encountered a solution by checking if it is implied by known solutions. This produces a large overhead that could easily be avoided by using an indexing-based approach instead.

In our experiments, we did not evaluate how our non-monotonic analysis performs compared to a monotonic analysis that uses the same abduction-based technique. To give an intuition, we can compare the *AST* approach with the *DFS* approach. The *DFS* approach can be seen as an improved monotonic analysis that first performs a sequence of strengthenings followed by a sequence of weakenings if it needs to backtrack. From Table 1, we can see that *AST* solves significantly more benchmarks than *DFS*. A purely monotonic analysis would still perform worse than *DFS*. However, we decided to not implement and compare monotonic approaches based on our abduction technique because the results could be too biased towards the non-monotonic analysis. In general, however, it is to expected that a non-monotonic analysis always has an edge over a monotonic analysis.

Threats to Validity. This is a very early experimental evaluation and therefore subject to several threats to validity. External threats to validity, that is the ability to generalize from our findings, are rooted in the selection of our benchmarks. Our abduction procedure depends to a large extend on the encoding of the horn clause problem. Hence, the heuristic in our A^* algorithm may need to be adjusted if a different encoding is used. We tried to address this issue by using benchmark sets from different sources but, ultimately, one can always find an example where our heuristic is particularly good or bad.

Internal threats to validity, that is in what way our experimental setup could be biased in favor of our approach, arise from the chosen timeout. To ensure that the selected timeout of 5 minutes is not biased towards our approach, we reran the experiments with timeouts of 10 and 15 minutes. Changing the timeout did not alter the number of benchmarks that can be solved by any of the tools.

6 Conclusion

We have presented a non-monotonic program analysis technique based on logical abduction. The approach bounds the non-monotonic search by a monotone sequence of checkpoints to enforce overall convergence. The benefit of using a non-monotonic analysis is that we can experiment with different search strategies to find a solution faster than with previous approaches that alternate monotonic pro-

gram analysis but do not interleave them. A central contribution of this paper is the A^*-like search presented in Section 4.3. Using a heuristic to determine how to alternate weakening and strengthening closely resembles the way a human would solve a set of horn clauses and opens an interesting space for future research.

Our experiments show that a heuristic search significantly outperforms breadth-first, and depth-first strategies. From these results, we can extrapolate that the heuristic search would also outperform a monotonic analysis that uses logical abduction. Although our current implementation cannot yet compete with established tools such as Z3 and Eldarica due to the preliminary implementation of the abduction engine, we demonstrated that non-monotonic search can solve problems that cannot be solved by these tools. We are confident that future improvements of the abduction engine and the search heuristic will result in a competitive and highly customizable tool.

References

[1] A. Albarghouthi, A. Gurfinkel, and M. Chechik. Whale: An interpolation-based algorithm for inter-procedural verification. In *VMCAI*, pages 39–55, 2012.

[2] T. A. Beyene, S. Chaudhuri, C. Popeea, and A. Rybalchenko. A constraint-based approach to solving games on infinite graphs. In *POPL*, pages 221–234. ACM, 2014.

[3] T. A. Beyene, C. Popeea, and A. Rybalchenko. Solving existentially quantified Horn clauses. In *CAV*, pages 869–882, 2013.

[4] N. Bjørner, K. McMillan, and A. Rybalchenko. Program verification as satisfiability modulo theories. In *SMT Workshop at IJCAR*, 2012.

[5] N. Bjørner, K. L. McMillan, and A. Rybalchenko. On solving universally quantified horn clauses. In *SAS*, pages 105–125, 2013.

[6] A. R. Bradley. Understanding IC3. In *Proc. SAT*, volume 7317 of *LNCS*, pages 1–14, 2012.

[7] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *SIGPLAN Not.*, 44(1):289–300, Jan. 2009.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th POPL*, 1977.

[9] P. Cousot and R. Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering*, 6(1):69–95, 1999.

[10] P. Cousot, R. Cousot, M. Fähndrich, and F. Logozzo. Automatic inference of necessary preconditions. In *VMCAI*, pages 128–148, 2013.

[11] I. Dillig and T. Dillig. Explain: A tool for performing abductive inference. In *Proceedings of the 25th International Conference on Computer Aided Verification*, CAV'13, pages 684–689, Berlin, Heidelberg, 2013. Springer-Verlag.

[12] I. Dillig, T. Dillig, and A. Aiken. Automated error diagnosis using abductive inference. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 181–192, New York, NY, USA, 2012. ACM.

[13] I. Dillig, T. Dillig, B. Li, and K. McMillan. Inductive invariant generation via abductive inference. *SIGPLAN Not.*, 48(10):443–456, Oct. 2013.

[14] E. Ermis, J. Hoenicke, and A. Podelski. Splitting via interpolants. In *VMCAI'12*, pages 186–201. Springer, 2012.

[15] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416, 2012.

[16] S. Gulwani and N. Jojic. Program verification as probabilistic inference. In *POPL*, 2007.

[17] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 235–246, New York, NY, USA, 2008. ACM.

[18] A. Gupta, C. Popeea, and A. Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In *POPL*, 2011.

[19] A. Gupta, C. Popeea, and A. Rybalchenko. Solving recursion-free Horn clauses over LI+UIF. In *APLAS*, pages 188–203, 2011.

[20] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, July 1968.

[21] M. Heizmann, J. Hoenicke, and A. Podelski. Nested interpolants. In *POPL*, 2010.

[22] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244. ACM, 2004.

[23] K. Hoder and N. Bjørner. Generalized property directed reachability. In *SAT*, 2012.

[24] K. R. M. Leino and F. Logozzo. Loop invariants on demand. In *APLAS*, volume 3780 of *LNCS*, pages 119–134, 2005.

[25] K. L. McMillan. Lazy abstraction with interpolants. In *CAV'06*, pages 123–136, 2006.

[26] K. L. McMillan and A. Rybalchenko. Solving constrained Horn clauses using interpolation. Technical Report MSR-TR-2013-6, Microsoft Research, 2013.

[27] M. Méndez-Lojo, J. A. Navas, and M. V. Hermenegildo. A flexible, (c)lp-based approach to the analysis of object-oriented programs. In *LOPSTR*, pages 154–168, 2007.

[28] Y. Moy. Sufficient preconditions for modular assertion checking. In *VMCAI*, pages 188–202, 2008.

[29] C. S. Pierce. *The Collected Papers of Charles Sanders Peirce*. Harvard University Press, 1935. Editors C. Hartshorne and P. Weiss.

[30] A. Podelski, A. Rybalchenko, and T. Wies. Heap assumptions on demand. In *CAV*, pages 314–327, 2008.

[31] P. Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In *Proceedings, 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 5330, pages 274–289, 2008.

[32] P. Rümmer, H. Hojjat, and V. Kuncak. Disjunctive interpolants for horn-clause verification. In *CAV*, pages 347–363, 2013.

[33] P. Rümmer and P. Subotic. Exploring interpolants. In *FMCAD*, pages 69–76, 2013.

[34] O. Sery, G. Fedyukovich, and N. Sharygina. Interpolation-based function summaries in bounded model checking. In *Haifa Verification Conference (HVC)*, Haifa, 2011. Springer.

Appendix

The results of our experiments are shown in Table 2. For each approach we record if it is able to solve a benchmark problem within 5 minutes. If so, we mark it with a ✓ and if it does not return a result within the time limit, we mark it with a ✗. Note that, for space reasons, we do not give the computation time for each approach and benchmark. Our approach is in most cases slower than Z3 and Eldarica for the examples that can be solved by both. This can be largely attributed to the prototypical implementation and the sub-optimal abduction procedure. We discuss our planned performance improvements below.

Benchmark	Z3	Eldarica	BFS	DFS	A^*
loop-01	✓	✓	✓	✓	✓
loop-02	✓	✓	✗	✗	✓
loop-03	✓	✓	✓	✓	✓
loop-04	✓	✓	✗	✓	✓
loop-04’	✓	✓	✗	✓	✓
loop-05	✓	✗	✓	✓	✓
loop-06	✓	✓	✗	✗	✓
loop-07	✗	✗	✗	✗	✗
loop-08	✓	✗	✗	✗	✓
loop-09	✓	✗	✗	✗	✗
loop-10	✓	✓	✗	✗	✗
loop-11	✓	✓	✗	✓	✓
loop-12	✓	✗	✓	✗	✗
loop-13	✓	✓	✗	✗	✗
loop-14	✗	✓	✗	✗	✗
loop-15	✗	✓	✗	✗	✗
loop-16	✗	✓	✗	✓	✓
loop-17	✓	✓	✗	✓	✓
loop-18	✗	✗	✗	✗	✗
loop-19	✓	✓	✓	✗	✓
loop-20	✓	✓	✗	✗	✗
loop-21	✗	✗	✗	✗	✗
loop-22	✓	✓	✗	✗	✗
loop-23	✓	✗	✓	✓	✓
loop-24	✓	✓	✓	✓	✓
loop-25	✓	✓	✓	✗	✓
loop-26	✓	✓	✗	✗	✗
loop-27	✓	✓	✓	✓	✓
loop-28	✓	✗	✓	✓	✓
loop-29	✓	✓	✓	✗	✓
loop-30	✗	✗	✓	✓	✓
loop-31	✓	✓	✓	✗	✓
loop-32	✗	✗	✗	✗	✗
loop-33	✓	✓	✗	✗	✓
loop-34	✓	✗	✗	✗	✗
loop-35	✓	✓	✗	✗	✗
loop-36	✓	✓	✗	✗	✗
loop-37	✓	✓	✗	✗	✗
loop-38	✗	✗	✗	✗	✗
loop-39	✓	✓	✗	✗	✗
loop-40	✓	✓	✗	✗	✓
loop-41	✓	✗	✓	✓	✓
loop-42	✓	✗	✗	✗	✗
loop-43	✓	✓	✓	✓	✓
loop-44	✗	✓	✗	✗	✗
loop-45	✓	✗	✗	✗	✗
loop-46	✓	✓	✓	✓	✓
RECUR-addition	✗	✓	✗	✗	✗
RECUR-bfprt	✓	✗	✗	✗	✗
RECUR-binarysearch	✓	✓	✓	✓	✓
RECUR-buildheap	✗	✗	✗	✗	✗
RECUR-countZero	✗	✓	✗	✗	✗
RECUR-floodfill	✓	✗	✗	✗	✗
RECUR-half	✗	✗	✗	✗	✗
RECUR-identity	✓	✓	✗	✗	✗
RECUR-mccarthy91	✓	✓	✗	✗	✓
RECUR-mccarthy92	✓	✓	✗	✗	✗
RECUR-merge	✗	✓	✗	✗	✗
RECUR-merge-leq	✗	✓	✗	✗	✗
RECUR-palindrome	✗	✓	✗	✗	✗
RECUR-parity	✓	✓	✗	✓	✓
RECUR-remainder	✗	✓	✗	✓	✓
RECUR-running	✗	✓	✗	✗	✗
RECUR-running-old	✓	✓	✓	✓	✓
RECUR-triple	✗	✓	✗	✗	✗
test/01	✗	✗	✓	✓	✓
test/04	✓	✓	✗	✓	✓
test/04x	✓	✓	✗	✓	✓
test/05	✗	✓	✗	✓	✓
test/11	✓	✓	✗	✗	✗
test/ackerman	✓	✓	✓	✓	✓
test/addition1	✗	✓	✗	✗	✗
test/cousot.correct	✗	✗	✗	✗	✗
test/dillig	✗	✗	✗	✗	✗
test/dillig2	✓	✓	✗	✗	✗
test/dillig-oopsla	✗	✗	✗	✗	✗
test/EvenOdd01	✓	✗	✓	✓	✓
test/fibonacci01	✓	✓	✗	✗	✗
test/fibonacci02	✓	✓	✗	✗	✗
test/hanoi	✗	✗	✗	✗	✗
test/mccarthy91	✓	✓	✗	✗	✓

Table 2: Results per benchmark. ✓means that a tool could solve the benchmark within 5 minute and ✗means it reached the timeout.