HoIce: An ICE-based Non-Linear Horn Clause Solver

Adrien Champion[1], Naoki Kobayashi[1], Ryosuke Sato[2]

[1] The University of Tokyo
[2] Kyushu University

Abstract. The ICE framework is a machine-learning-based technique originally introduced for inductive invariant inference over transition systems, and building on the supervised learning paradigm. Recently, we adapted the approach to non-linear Horn clause solving in the context of higher-order program verification. We showed that we could solve more of our benchmarks (extracted from higher-order program verification problems) than other state-of-the-art Horn clause solvers. This paper discusses some of the many improvements we recently implented in HoIce, our implementation of this generalized ICE framework.

1 Introduction

Constrained Horn clauses is a popular formalism for encoding program verification problems [4–6], and efficient Horn clause solvers have been developed over the last decade [3, 9, 10]. Recently, we adapted the Ice framework [7, 8] to non-linear Horn clause solving [6]. Our experimental evaluation on benchmarks encoding the verification of higher-order functional programs as (non-linear) Horn clauses showed that our generalized Ice framework outperformed existing solvers in terms of precision. This paper discusses HoIce[3], a Horn clause solver written in Rust [1] implementing the generalized Ice framework from [6]. Let us briefly introduce Horn clause solving before presenting HoIce in more details.

Given a set of unknown predicates Π, a *(constrained) Horn clause* is a constraint of the form

$$\forall v_0, \ldots, v_n \mid \Phi \ \wedge \bigwedge_{i \in I} \{\pi_i(\vec{a}_i)\} \ \models \ H$$

where Φ is a formula and each $\pi_i(\vec{a}_i)$ is an application of $\pi_i \in \Pi$ to some arguments $\vec{a}_i$. The *head* of the clause H is either the formula *false* (written $\bot$) or a predicate application $\pi(\vec{a})$. Last, $v_0, \ldots, v_n$ are the free variables appearing in Φ, the predicate applications and H. We follow tradition and omit the quantification over $v_0, \ldots, v_n$ in the rest of the paper. To save space, we will occasionally write $\langle \Phi, \ \{\pi_i(\vec{a}_i)\}_{i \in I}, \ H \rangle$ for the clause above.

A set of Horn clauses is *satisfiable* if there exist definitions for the predicates in Π that verify all the Horn clauses. Otherwise, it is *unsatisfiable*. A Horn clause solver implements a decision procedure for Horn clauses satisfiability. A solver is also usually expected to be able to yield some definitions of the predicates, when the Horn clauses are satisfiable.

[3] Available at https://github.com/hopv/hoice

Example 1. Let $\Pi = \{\pi\}$ and consider the following Horn clauses:

$$n > 100 \models \pi(n, n-10) \quad (1)$$
$$\neg(n > 100) \wedge \pi(n+11, tmp) \wedge \pi(tmp, res) \models \pi(n, res) \quad (2)$$
$$m \leq 101 \wedge \neg(res = 91) \wedge \pi(m, res) \models \bot \quad (3)$$

These Horn clauses are satisfiable, for instance with

$$\pi(n, res) \equiv (res = 91) \vee (n > 101 \wedge res = n - 10).$$

Section 2 describes a use-case for Horn clause solving and briefly discusses HoIce's interface. Section 3 provides a succinct description of the generalized Ice framework HoIce relies on. In Section 4 we discuss the most important improvements we implemented in HoIce since v1.0.0 [6] for the v1.5.0 release. Next, Section 5 evaluates HoIce on our set of benchmarks stemming from higher-order program verification problems, as well as all the benchmarks submitted to the first CHC-COMP Horn clause competition[4] in the linear integer or linear real arithmetic fragments. Finally, Section 6 discusses future work.

2 Applications and Interface

As mentioned above, Horn clauses is a popular and well-established formalism to encode program verification, especially imperative program verification [4–6]. HoIce however is developed with (higher-order) functional program verification in mind, in particular through refinement/intersection type inference. We thus give an example of using Horn clauses for refinement type inference.

Example 2. Consider the program using McCarthy's 91 function below (borrowed from [6]). We are interested in proving the assertion in `main` can never fail.

```
let rec mc_91 n = if n > 100 then n - 10
                  else let tmp = mc_91 (n + 11) in mc_91 tmp
let main m = let res = mc_91 m in if m ≤ 101 then assert (res = 91)
```

To prove this program is safe, it is enough to find a predicate π such that `mc_91` has (refinement) type $\{n : \texttt{int} \mid true\} \to \{res : \texttt{int} \mid \pi(n, res)\}$ and π satisfies $\forall m, res \mid m \leq 101 \wedge \neg(res = 91) \wedge \pi(n, res) \models \bot$.

The latter is already a Horn clause, and is actually (3) from Example 1. Regarding the constraints for (refinement) typing `mc_91`, we have to consider the two branches of the conditional statement in its definition. The first branch yields clause (1). The second one yields clause (2), where *res* corresponds to the result of `mc_91` *tmp*.

Horn clause solvers are typically used by program verification tools. Such tools handle the high-level task of encoding the safety of a given program as Horn clauses. The clauses are passed to the solver and the result is communicated back through

[4] https://chc-comp.github.io/

library calls, process input/output interaction, or files. This is the case, for instance, of r_type [6], which encodes refinement type inference as illustrated in Example 2. It then passes the clauses to HoIce, and rewrites the Horn-clause-level result in terms of the original program. Communication with HoIce is (for now) strictly text-based: either interactively by printing (reading) on its standard input (output), or by passing a file. We give a full example of the SMT-LIB-based [2] input language of HoIce in Appendix A, and refer the reader to Appendix B for a partial description of HoIce's arguments.

3 Generalized Ice

This section provides a quick overview of the generalized Ice framework HoIce is based on. We introduce only the notions we need to discuss, in Section 4 the improvements we have recently implemented. Ice, both the original and generalized versions, are supervised learning frameworks, meaning that they consist of a teacher and a learner. The latter is responsible for producing candidate definitions for the predicates to infer, based on ever-growing learning data (defined below) provided by the teacher. The teacher, given some candidates from the learner, checks whether they respect the Horn clauses, typically using an SMT solver[5]. If they do not, the teacher asks for a new candidate after generating more learning data. We are in particular interested in the generation of learning data, discussed below after we introduce Horn clause traits of interest.

A Horn clause $\langle \Phi,\ \{\pi_i(\vec{a}_i)\}_{i\in I},\ H\rangle$ is *positive* if $I = \emptyset$ and $H \neq \bot$, *negative* if $I \neq \emptyset$ and $H = \bot$, and is called an *implication* clause otherwise. A negative clause is *strict* if $|I| = 1$, and *non-strict* otherwise. For all $\pi \in \Pi$, let $C(\pi)$ be the candidate provided by the learner. A *counterexample* for a Horn clause $\langle \Phi,\ \{\pi_i(\vec{a}_i)\}_{i\in I},\ H\rangle$ is a model for

$$\neg(\ \Phi \wedge \bigwedge_{i\in I} C(\pi_i)(\vec{a}_i)\ \Rightarrow\ C(H)\ ),$$

where $C(H)$ is $C(\pi)(\vec{a})$ if H is $\pi(\vec{a})$ and $\bot$ otherwise.

A *sample* for $\pi \in \Pi$ is a tuple of concrete values $\vec{v}$ for its arguments, written $\pi(\vec{v})$. Samples are generated from Horn clause counterexamples, by retrieving the value of the arguments of the clause's predicate applications. The generalized Ice framework maintains *learning data* made of (collections of) samples extracted from Horn clause counterexamples. There are three kinds of learning data depending on the shape of the falsifiable clause.

From a counterexample for a positive clause, the teacher extracts a *positive sample*: a single sample $\pi(\vec{v})$, encoding that π must evaluate to true on $\vec{v}$. A counterexample for a negative clause yields a *negative constraint*: a set of samples $\{\pi_i(\vec{v}_i)\}_{i\in I}$ encoding that there must be at least one $i \in I$ such that π_i evaluates to false on $\vec{v}_i$. We say a negative constraint is a *negative sample* if it is a singleton set. An *implication constraint* is a pair $(\ \{\pi_i(\vec{v}_i)\}_{i\in I},\ \pi(\vec{v})\ )$ and comes from a counterexample to an implication clause. Its semantics is that if all $\pi_i(\vec{v}_i)$ evaluate to true, $\pi(\vec{v})$ must evaluate to true.

[5] HoIce uses the Z3 [12] SMT solver.

Example 3. Say the current candidate is $\pi(v_0, v_1) \equiv \bot$, then (1) is falsifiable and yields, for instance, the positive sample $\pi(101, 91)$. Say now the candidate is $\pi(v_0, v_1) \equiv v_0 = 101$. Then (3) is falsifiable and it might yield the negative sample $\pi(101, 0)$. Last, (2) is also falsifiable and can generate the constraint $(\{\pi(101, 101), \pi(101, 0)\},\ \pi(101, 0))$.

We do not discuss in details how the learner generates candidates here and instead highlight its most important features. First, when given some learning data, the learner generates candidates that respect the semantics of all positive samples and implication/negative constraints. Second, the learner has some freedom in how it respect the constraints. Positive/negative samples are *classified* samples in the sense that they force some predicate to be true/false for some inputs. Constraints on the other hand contain *unclassified* samples, meaning that the learner can, to some extent, decide whether the candidates it generates evaluate to true or false on these samples.

4 Improvements

We invested a lot of efforts to improve HoIce since v1.0.0. Besides bug fixes and all-around improvements, HoIce now supports the theories of reals and arrays, as opposed to integers and booleans only previously. The rest of this section presents the improvements which, according to our experiments, are the most beneficial in terms of speed and precision. The first technique extends the notion of sample to capture more than one samples at the same time, while Section 4.2 aims at producing more positive/negative samples to better guide the choices in the learning process.

4.1 Partial Samples

Most modern SMT-solvers are able to provide extremely valuable information in the form of *partial models*. By omitting some of the variables when asked for a model, they communicate the fact that the values of these variables are irrelevant (given the values of the other variables). In our context, this information is extremely valuable.

Whenever the teacher retrieves a counterexample for a clause where some variables are omitted, it can generate partial learning data composed of samples where values can be omitted. Each partial sample thus covers many complete samples, infinitely many if the variable's domain is infinite. This of course assumes that the learner is able to handle such partial samples, but in the case of the decision-tree-based approach introduced in [8] and generalized in [6], supporting partial samples is straightforward. Typically, one discards all the qualifiers that mention at least one of the unspecified variables, and proceeds with the remaining ones following the original qualifier selection approach.

4.2 Constraint Breaking

This section deals with the generation of learning data in the teacher part of the ICE framework. Given some candidates, our goal is to generate data *i)* refuting the current candidate, and *ii)* the learner will have few (classification) choices to make about.

In the rest of this section, assume that the teacher is working on clause $\langle \Phi,\ \{\pi_i(\vec{a}_i)\}_{i\in I},\ H\rangle$, which is falsifiable *w.r.t.* the current candidate C. Assume also that this clause is either

an implication clause or a non-strict negative clause. This means that the teacher will generate either an implication constraint or a non-strict negative one, meaning that the learner will have to classify the samples appearing in these constraints. We are interested in *breaking* these constraints to obtain positive or strict negative samples at best, and smaller constraints at worst. If we can do so, the learner will have fewer choices to make to produce a new candidate. Let us illustrate this idea on an example.

Example 4. Assume that our generalized Ice framework is working on the clauses from Example 1. Assume also that the learning data only consists of positive sample $\pi(101, 91)$, and the current candidate is $\pi(v, v') \equiv v \geq 101 \wedge v' = v - 10$. Implication clause (2) $\langle \neg(n > 100),\ \{\pi(n + 11, tmp),\ \pi(tmp, res)\},\ \pi(n, res)\rangle$ is falsifiable. Can we force one of the predicate applications in the set to be our positive sample? It turns out $\pi(tmp, res)$ can, yielding constraint $(\ \{\pi(111, 101),\ \pi(101, 91)\},\ \pi(100, 91)\ )$, which is really $(\ \{\pi(111, 101)\},\ \pi(100, 91)\ )$ since we know $\pi(101, 91)$ must be true.
We could simplify this constraint further if we had $\pi(111, 101)$ as a positive sample. It is indeed safe to add it as a positive sample because it can be obtained from clause (1) by checking whether $n > 100 \ \wedge \ n = 111 \ \wedge \ (n - 10) = 101$ is satisfiable, which it is. So, instead of generating an implication constraint mentioning three samples the learner would have to make choices on, we ended up generating two new positive samples $\pi(111, 101)$ and $\pi(100, 91)$. (The second sample is the one rejecting the current candidate.)

The rest of this section presents two techniques we implemented to accomplish this goal. The first one takes place during counterexample extraction, while the second one acts right after the extraction. In the following, for all $\pi \in \Pi$, let $\mathcal{P}(\pi)$ (*resp.* $\mathcal{N}(\pi)$) be the positive (*resp.* negative) samples for π. $C(\pi)$ refers to the current candidate for π, and by extension $C(H)$ for the head H of a clause is $C(\pi)(\vec{a})$ if H is $\pi(\vec{a})$ and $\bot$ otherwise.

Improved Counterexample Extraction This first approach consists in forcing some arguments for a predicate application of π to be in $\mathcal{P}(\pi)$ or $\mathcal{N}(\pi)$. This means that we are interested in models of the following satisfiable formula:

$$\Phi \wedge \bigwedge_{i \in I} \{C(\pi_i)(\vec{a}_i)\} \ \wedge \ \neg C(H)(\vec{a}). \tag{4}$$

Positive Reduction. Assume that H is $\pi(\vec{a})$. Let $I_+ \subseteq I$ be the indexes of the predicate applications that can individually be forced to a known positive sample; more formally, $i \in I_+$ if and only if the conjunction of (4) and $P_i \equiv \bigvee_{\vec{v} \in \mathcal{P}(\pi_i)} (\vec{a}_i = \vec{v})$ is satisfiable. Then, if $I_+ \neq \emptyset$ and the conjunction of (4) and $\bigwedge_{i \in I_+} P_i$ is satisfiable, a model for this conjunction refutes the current candidate and yields a smaller constraint than a model for (4) alone would. (This technique was used in the first simplification of Example 4.)

Negative Reduction. Let N be $\bigvee_{\vec{v} \in \mathcal{N}(\pi)} (\vec{a} = \vec{v})$ if H is $\pi(\vec{a})$, and true if H is $\bot$. Assuming $I_+ \neq \emptyset$, we distinguish two cases. If $I_+ = I$, then for all $j \in I_+$, if (4) and N and $\bigwedge_{i \in I_+,\ i \neq j} P_i$ is satisfiable, a model for this conjunction yields a strict negative sample for π_j. Otherwise, if (4) and N and $\bigwedge_{i \in I_+} P_i$ is satisfiable, a model for this conjunction yields a negative sample mentioning the predicates in $I \setminus I_+$.

Post-Extraction Simplification This second technique applies to implication and non-strict negative constraints right after they are generated from the counterexamples for a candidate. Let us define the predicate $isPos(\pi, \vec{v})$ for all $\pi \in \Pi$, where $\vec{v}$ are concrete input values for π. This predicate is true if and only if there is a positive clause $\langle \Phi,\ \emptyset,\ \pi(\vec{a}) \rangle$ such that $\Phi \wedge (\vec{a} = \vec{v})$ is satisfiable. Likewise, let $isNeg(\pi, \vec{v})$ be true if and only if there is a strict negative clause $\langle \Phi,\ \{\pi(\vec{a})\},\ \bot \rangle$ such that $\Phi \wedge (\vec{a} = \vec{v})$ is satisfiable.

Now we can go through the samples appearing in the constraints and check whether we can infer that they should be positive or negative using *isPos* and *isNeg*. This allows to both discover positive/negative samples, and simplify constraints so that the learner has fewer choices to make. (This technique was used in the second simplification step in Example 4.) Notice in particular that discovering a negative (positive) sample in non-strict negative data or in the antecedents of implication data (consequent of implication data) breaks it completely.

5 Evaluation

We now evaluate the improvements discussed in Section 4. The benchmarks we used consist of all 3586 benchmarks submitted to the CHC-COMP 2018[6] that use only booleans and linear integer or real arithmetic. We did not consider benchmarks using arrays as their treatment in the learner part of HoIce is currently quite naïve.

Figure 1 compares HoIce 1.0 with different variations of HoIce 1.5 where the techniques from Section 4 are activated on top of one another. That is, "hoice inactive" has none of them active, "hoice partial" activates partial samples (Section 4.1), and "hoice breaking" activates partial samples *and* constraint breaking (Section 4.2). We discuss the exact options used in Appendix B.

We first note that even without the improvements discussed in Section 4, HoIce 1.5 is significantly better than HoIce 1.0 thanks to the many optimizations, tweaks and new minor features implemented since then. Next, the huge gain in precision and speed thanks to partial samples cannot be overstated: partial samples allow the framework to represent an infinity of samples with a single one by leveraging information that comes for free from the SMT-solver. Constraint breaking on the other hand does not yield nearly as big an improvement. It was implemented relatively recently and a deeper analysis on how it affects the generalized Ice framework is required to draw further conclusions.

Next, let us evaluate HoIce 1.5 against the state of the art Horn clause solver Spacer [11] built inside Z3 [12]. We used the latest revision of Z3 at the time of writing: `d6298b0`. Figure 2a shows a comparison on our benchmarks [7] stemming from higher-order functional programs. The timeout is 30 seconds, and the solvers are asked to produce definitions which are then verified. The rational behind checking the definitions is that in the context of refinement/intersection type inference, the challenge is to produce types for the function that ensure the program is correct. The definitions are thus important for us, since the program verification tool using HoIce in this context will ask

[6] https://chc-comp.github.io/

[7] Available at https://github.com/hopv/benchmarks/tree/master/clauses

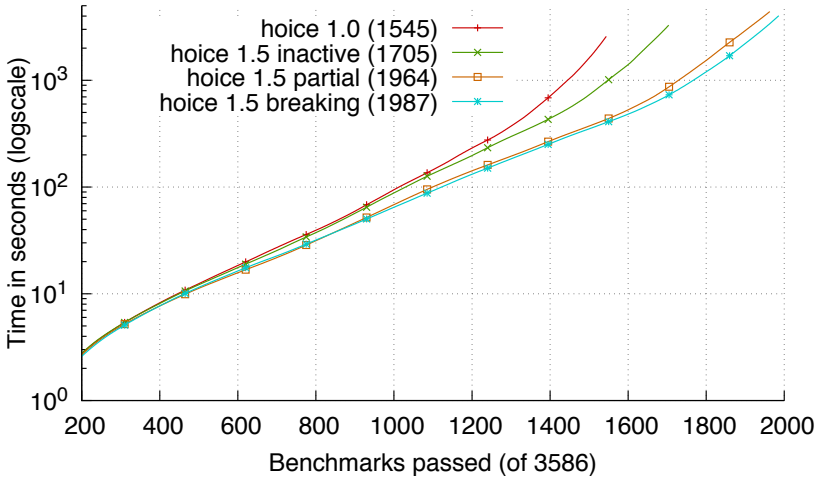
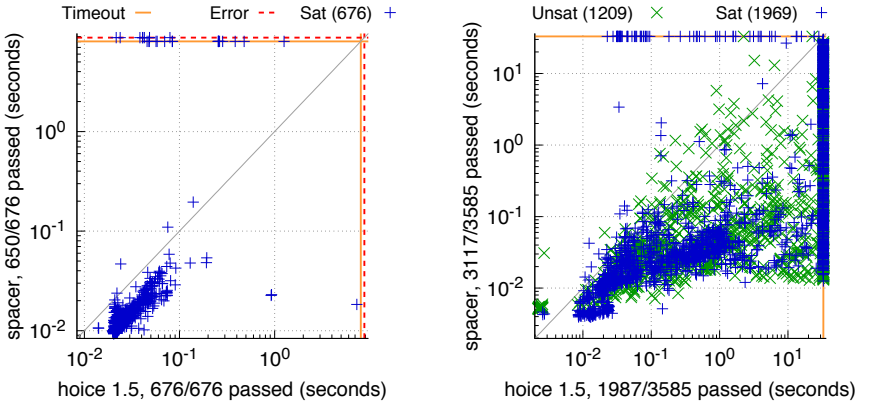


Fig. 1: Cumulative plot over the CHC-COMP 2018 linear arithmetic benchmarks.

for them. Spacer clearly outperforms HoIce on the benchmarks it can solve, but fails on 26 of them. While 17 are actual timeouts, Spacer produces definitions that do not verify the clauses on the remaining 9 benchmarks. The problem has been reported but is not resolved at the time of writing. Regardless of spurious definitions, HoIce still proves more (all) of our benchmarks.

Last, Figure 2b compares HoIce and Spacer on the CHC-COMP benchmarks mentioned above. A lot of them are large enough that checking the definitions of the predicates is a difficult problem in itself: we thus did not check the definitions for these benchmarks for practical reasons. There are 632 satisfiable (438 unsatisfiable) benchmarks that Spacer can solve on which HoIce reaches the timeout, and 49 satisfiable (4 unsatisfiable) that HoIce can solve but Spacer times out on. Spacer is in general much faster and solves a number of benchmarks much higher than HoIce. We see several reasons for this. First, some of the benchmarks are very large and trigger bottlenecks in HoIce, which is a very young tool compared to Z3/Spacer. These are problems of the implementation (not of the approach) that we are currently addressing. Second, HoIce is optimized for solving clauses stemming from functional program verification. The vast majority of the CHC-COMP benchmarks come from imperative program verification, putting HoIce out of its comfort zone. Last, a lot of these benchmarks are unsatisfiable, which the ICE framework in general is not very good at. HoIce was developed completely for satisfiable Horn clauses, as we believe proving unsatisfiability (proving programs unsafe) would be better done by a separate engine. Typically a bounded model-checking tool.



(a) On our benchmarks. **(b)** On the CHC-COMP 2018 benchmarks.

Fig. 2: Comparison between HoIce and Z3 Spacer.

6 Conclusion

In this paper we discussed the main improvements implemented in HoIce since version 1.0. We showed that the current version outperforms Spacer on our benchmarks stemming from higher-order program verification.

Besides the never-ending work on optimizations and bug fixes, our next goal is to support the theory of Algebraic Data Types (ADT). In our context of higher-order functional program verification, it is difficult to find interesting, realistic use-cases that do not use ADTs.

Acknowledgments

We thank the anonymous referees for useful comments. This work was supported by JSPS KAKENHI Grant Number JP15H05706.

References

1. The Rust language. https://www.rust-lang.org/en-US/
2. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org (2016)
3. Bjørner, N., Gurfinkel, A., McMillan, K.L., 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 - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday. Lecture Notes in Computer Science, vol. 9300, pp. 24–51. Springer (2015)
4. Bjørner, N., McMillan, K.L., Rybalchenko, A.: Program verification as satisfiability modulo theories. In: SMT@IJCAR. EPiC Series in Computing, vol. 20, pp. 3–11. EasyChair (2012)
5. Bjørner, N., McMillan, K.L., Rybalchenko, A.: Higher-order program verification as satisfiability modulo theories with algebraic data-types. CoRR abs/1306.5264 (2013)

6. Champion, A., Chiba, T., Kobayashi, N., Sato, R.: Ice-based refinement type discovery for higher-order functional programs. In: TACAS (1). Lecture Notes in Computer Science, vol. 10805, pp. 365–384. Springer (2018)
7. Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: A robust framework for learning invariants. In: Proceedings of CAV 2014. LNCS, vol. 8559, pp. 69–87. Springer (2014)
8. Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. In: Proceedings of POPL 2016. pp. 499–512. ACM (2016)
9. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Cimatti, A., Sebastiani, R. (eds.) Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7317, pp. 157–171. Springer (2012)
10. Hojjat, H., Konecný, F., Garnier, F., Iosif, R., Kuncak, V., Rümmer, P.: A verification toolkit for numerical transition systems - tool paper. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7436, pp. 247–251. Springer (2012)
11. Komuravelli, A., Gurfinkel, A., Chaki, S., Clarke, E.M.: Automatic abstraction in smt-based unbounded software model checking. CoRR abs/1306.1945 (2013)
12. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Proceedings of TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer (2008)

```
(set-logic HORN)

(declare-fun mc_91 ( Int Int ) Bool)

(assert
  (forall ( (n Int) )
    (=> (> n 100)
        (mc_91 n (- n 10))
) ) )
(assert
  (forall ( (n Int) (tmp Int) (res Int) )
    (=> (and (not (> n 100)) (mc_91 (+ n 11) tmp) (mc_91 tmp res))
        (mc_91 n res)
) ) )
(assert
  (forall ( (m Int) (res Int) )
    (=> (and (<= m 101) (mc_91 m res))
        (= res 91)
) ) )

(check-sat)
(get-model)
```

Fig. 3: A legal input script corresponding to Example 1.

A Input/Output Format Example

This section illustrates HoIce's input/output format. For a complete discussion on the format, please refer to the HoIce wiki https://github.com/hopv/hoice/wiki. HoIce takes special SMT-LIB [2] scripts as inputs such as the one on Figure 3. A script starts with an optional `set-logic HORN` command, followed by some predicate declarations using the `declare-fun` command. Only predicate declaration are allowed: all declarations must have codomain `Bool`.

The actual clauses are given as assertions which generally start with some universally quantified variables, wrapping the implication between the body and the head of the clause. Negated existential quantification is also supported, for instance the third assertion on Figure 3 can be written as

```
(assert
  (not
    (exist ( (m Int) (res Int) )
      (and (<= m 101) (mc_91 m res) (not (= res 91)))
) ) )
```

The `check-sat` command asks whether the Horn clauses are satisfiable, which they are, and HoIce answers `sat`. Otherwise, it would have answered `unsat`. Since the clauses are satisifiable, it is legal to ask for a model using the `get-model` command. HoIce provides one in the standard SMT-LIB fashion:

```
(model
  (define-fun mc_91
    ( (v_0 Int) (v_1 Int) ) Bool
    (or
      (and (= (+ v_0 (- 10) (* (- 1) v_1)) 0) (or (= (+ v_1 (- 91)) 0) (>= v_0 102)))
      (and (>= (* (- 1) v_0) (- 100)) (or (= (+ v_1 (- 91)) 0) (>= v_0 102))
           (not (= (+ v_0 (- 10) (* (- 1) v_1)) 0))
) ) ) )
```

Note that hoice can read scripts from files, but also on its standard input in an interactive manner.

B Arguments

HoIce has no mandatory arguments. Besides options and flags, users can provide a file path argument in which case HoIce reads the file as an SMT-LIB script encoding a Horn clause problem (see Appendix A). When called with no file path argument, HoIce reads the script from its standard input. In both cases, HoIce outputs the result on its standard output.

Running HoIce with `-h` or `--help` will display the (visible) options. We do not discuss them here. Instead, let us clarify which options we used for the results presented in Section 5. The relevant option for partial samples from Section 4.1 is `--partial`, while `--bias_cexs` and `--assistant` activate constraint breaking as discussed in Section 4.2. More precisely, `--bias_cexs` activates contraint breaking during counterexample extraction, while `--assistant` triggers post-extraction simplification. The commands ran for the variants of Figure 1 are thus

hoice 1.5 inactive	`hoice --partial off --bias_cexs off --assistant off`
hoice 1.5 partial	`hoice --partial on --bias_cexs off --assistant off`
hoice 1.5 breaking	`hoice --partial on --bias_cexs on --assistant on`

As far as the experiments are concerned, we ran Z3 revision `d6298b0` with only one option, the one activating Spacer: `fixedpoint.engine=spacer`.