Classifying and Solving Horn Clauses for Verification

Philipp Rümmer[1], Hossein Hojjat[2], and Viktor Kuncak[2]

[1] Uppsala University, Sweden
[2] Swiss Federal Institute of Technology Lausanne (EPFL)

Abstract. As a promising direction to overcome difficulties of verification, researchers have recently proposed the use of Horn constraints as intermediate representation. Horn constraints are related to Craig interpolation, which is one of the main techniques used to construct and refine abstractions in verification, and to synthesise inductive loop invariants. We give a classification of the different forms of Craig interpolation problems found in literature, and show that all of them correspond to natural fragments of (recursion-free) Horn constraints. For a logic that has the binary interpolation property, all of these problems are solvable, but have different complexity. In addition to presenting the theoretical classification and solvability results, we present a publicly available collection of benchmarks to evaluate solvers for Horn constraints, categorized according to our classification. The benchmarks are derived from real-world verification problems. The behavior with our tools as well as with Z3 prover indicates the importance of Horn clause solving as distinct from the general problem of solving quantified constraints by quantifier instantiation.

1 Introduction

Predicate abstraction [14] has emerged as a prominent and effective way for model checking software systems. A key ingredient in predicate abstraction is analyzing the spurious counter-examples to refine abstractions [4]. The refinement problem saw a significant progress when Craig interpolants extracted from unsatisfiability proofs were used as relevant predicates [19]. While interpolation has enjoyed a significant progress for various logical constraints [7–9, 23], there have been substantial proposals for more general forms of interpolation [1, 18, 23].

As a promising direction to extend the reach of automated verification methods to programs with procedures, and concurrent programs, among others, recently the use of Horn constraints as intermediate representation has been proposed [15, 16, 27]. This paper examines the relationship between various forms of Craig interpolation and syntactically defined fragments of recursion-free Horn clauses. We systematically examine binary interpolation, inductive interpolant sequences, tree interpolants, restricted DAG interpolants, and disjunctive interpolants, and show the recursion-free Horn clause problems to which they correspond. We present algorithms for solving each of these classes of problems by reduction to elementary interpolation problems. We also give a taxonomy of the various interpolation problems, and the corresponding systems of Horn clauses, in terms of their computational complexity.

The contributions of the paper are:

- a systematic study of relevant recursion-free Horn fragments, their relationship to forms of Craig interpolation, and their computational complexity;
- a library of recursion-free Horn problems, designed for benchmarking Horn solvers and interpolation engines;
- the generalisation of our results from recursion-free Horn clauses to general well-founded constraints, i.e., to constraints without infinite resolution proofs.

Organisation. Related work is surveyed in Sect. 2, following in Sect. 3 by an example of (recursive) Horn clauses. Sect. 4 formally introduces the concept of Horn clauses. Sect. 5 investigates the relationship between Horn fragments and Craig interpolation, and Sect. 6 their respective computational complexity. Sect. 7 presents our library of Horn benchmarks. Sect. 8 generalises from Horn clauses to well-founded clauses.

2 Related Work

The use of **Horn clauses** as intermediate representation for verification was proposed in [28]. [16] uses Horn clauses for verification of multi-threaded programs. The underlying procedure for solving sets of recursion-free Horn clauses, over the combined theory of linear integer arithmetic and uninterpreted functions, was presented in [17]. A range of further applications of Horn clauses, including inter-procedural model checking, was given in [15]. Horn clauses are also proposed as intermediate/exchange format for verification problems in [6], and are natively supported by the SMT solver Z3 [11].

There is a long line of research on **Craig interpolation** methods, and generalised forms of interpolation, tailored to verification. For an overview of interpolation in the presence of theories, we refer the reader to [8, 9]. Binary Craig interpolation for implications $A \rightarrow C$ goes back to [10], was carried over to conjunctions $A \wedge B$ in [24], and generalised to inductive sequences of interpolants in [19, 26]. The concept of tree interpolation, strictly generalising inductive sequences of interpolants, is presented in the documentation of the interpolation engine iZ3 [23]; the computation of tree interpolants by computing a sequence of binary interpolants is also described in [18]. Restricted DAG interpolants [1] and disjunctive interpolants [29] are further generalisations of inductive sequences of interpolants, designed to enable the simultaneous analysis of multiple counterexamples or program paths.

The use of Craig interpolation for solving Horn clauses is discussed in [27], concentrating on the case of tree interpolation. Our paper extends this work by giving a systematic study of the relationship between different forms of Craig interpolation and Horn clauses, as well as general results about solvability and computational complexity, independent of any particular calculus used to perform interpolation.

Inter-procedural software model checking with interpolants has been an active area of research for the last decade. In the context of predicate abstraction, it has been discussed how well-scoped invariants can be inferred [19] 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]. Generalisations of the Impact algorithm [26] to programs with procedures are given in [18] (formulated using nested word automata) and [2]. Finally, function summaries generated using interpolants have also been used to speed up bounded model checking [30].

```
def f(n : Int)
      returns rec : Int = {
  if (n > 0) {
    tmp = f(n−1)
    rec = tmp + 1
  } else {
    rec = 1
  }
}
def main() {
  var res : Int
  havoc(x: Int ≥ 0)
  res = f(x)
  assert(res == x + 1)
}
```

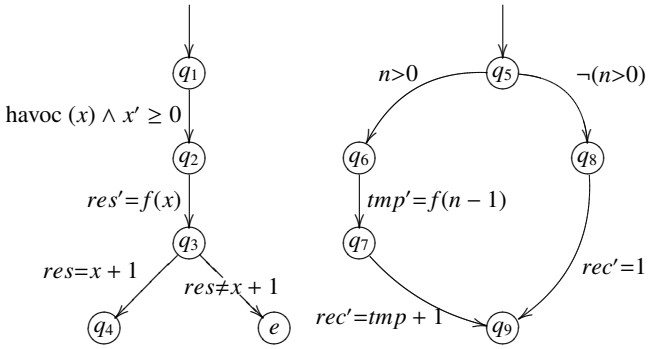


Fig. 1. A recursive program and its control flow graph (see Sect. 3).

(1) r1(X, Res) ← **true**
(2) r2(X', Res) ← r1(X, Res) ∧ X' ≥ 0
(3) r3(X, Res') ← r2(X, Res) ∧ rf(X, Res')
(4) r4(X, Res) ← r3(X, Res) ∧ Res = X + 1
(5) **false** ← r3(X, Res) ∧ Res ≠ X + 1

(6) r5(N, Rec, Tmp) ← **true**
(7) r6(N, Rec, Tmp) ← r5(N, Rec, Tmp) ∧ N > 0
(8) r7(N, Rec, Tmp') ← r6(N, Rec, Tmp) ∧ rf(N − 1, Tmp')
(9) r8(N, Rec, Tmp) ← r5(N, Rec, Tmp) ∧ N ≤ 0
(10) r9(N, Rec', Tmp) ← r7(N, Rec, Tmp) ∧ Rec' = Tmp + 1
(11) r9(N, Rec', Tmp) ← r8(N, Rec, Tmp) ∧ Rec' = 1
(12) rf(N, Rec) ← r9(N, Rec, Tmp)

Fig. 2. The encoding of the program in Fig. 1 into a set of recursive Horn clauses.

Several other tools handle procedures by increasingly inlining and performing under and/or over-approximation [21,31,32], but without the use of interpolation techniques.

3 Example

We start with an example illustrating the use of Horn clauses to verify a recursive program. Fig. 1 shows an example of a recursive program, which is encoded as a set of (recursive) Horn constraints in Fig. 2. The function `f` recursively computes the increment of the argument `n` by 1.

For translation to Horn clauses we assign an uninterpreted relation symbol `r`i to each state q_i of the control flow graph. The arguments of the relation symbol `r`i act as placeholders of the visible variables in the state q_i. The relation symbol `rf` corresponds

$$
\begin{aligned}
r_1(x, res) &\equiv true \\
r_2(x, res) &\equiv x \geq 0 \\
r_3(x, res) &\equiv res = x + 1 \\
r_4(x, res) &\equiv true \\
r_5(n, rec, tmp) &\equiv true \\
r_6(n, rec, tmp) &\equiv n \geq 1 \\
r_7(n, rec, tmp) &\equiv n = tmp \\
r_8(n, rec, tmp) &\equiv n \leq 0 \\
r_9(n, rec, tmp) &\equiv rec = n + 1 \vee (n \leq 0 \wedge rec = 1) \\
r_f(n, rec) &\equiv rec = n + 1 \vee (n \leq 0 \wedge rec = 1)
\end{aligned}
$$

Fig. 3. Syntactic solution of the Horn clauses in Fig. 2.

to the summary of the function `f`. In the relation symbol `rf` we do not include the local variable `tmp` in the arguments since it is invisible from outside the function `f`. The first argument of `rf` is the input and the second one is the output. We do not dedicate any relation symbol to the error state e.

The initial states of the functions are not constrained at the beginning; they are just implied by *true*. The clause that has *false* as its head corresponds to the assertion in the program. In order to satisfy the assertion with the head *false*, the body of the clause should also be evaluated to *false*. We put the condition leading to error in the body of this clause to ensure the error condition is not happening. The rest of the clauses are one to one translation of the edges in the control flow graph.

For the edges with no function calls we merely relate the variables in the previous state to the variables in the next state using the transfer functions on the edges. For example, the clause (2) expresses that *res* is kept unchanged in the transition from q_1 to q_2 and the value of `x` is greater than or equal to 0 in q_2. For the edges with function call we should also take care of the passing arguments and the return values. For example, the clause (3) corresponds to the edge containing a function call from q_2 to q_3. This clause sets the value of `res` in the state q_3 to the return value of the function `f`. Note that the only clauses in this example that have more than one relation symbols in the body are the ones related to edges with function calls.

The solution of the obtained system of Horn clauses demonstrates the correctness of the program. In a solution each relation symbol is mapped to an expression over its arguments. If we replace the relation symbols in the clauses by the expressions in the solution we should obtain only valid clauses. In a system with a genuine path to error we cannot find any solution to the system since we have no way to satisfy the assertion clause. Fig. 3 gives one possible solution of the Horn clauses in terms of concrete formulae, found by our verification tool Eldarica.[3]

[3] http://lara.epfl.ch/w/eldarica

This paper discusses techniques to automatically construct solutions of Horn clauses. Although the Horn clauses encoding programs are typically recursive, it has been observed that the case of *recursion-free* Horn clauses is instrumental for constructing verification procedures operating on Horn clauses [15, 16, 27]. Sets of recursion-free Horn clauses are usually extracted from recursive clauses by means of finite unwinding; examples are given in Sect. 5.3 and 5.5.

4 Formulae and Horn Clauses

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

A constraint is called *satisfiable* if it holds for some structure in $\mathcal{S}$ and some assignment of the variables $\mathcal{X}$, 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 $\Gamma \models \phi$.

$fv(\phi)$ denotes the set of free variables in constraint 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 containing the free variables $x_1, \ldots, x_n$, we write $Cl_\forall(\phi)$ for the *universal closure* $\forall x_1, \ldots, x_n.\phi$.

Craig interpolation is the main technique used to construct and refine abstractions in software model checking. A binary interpolation problem is a conjunction $A \wedge B$ of constraints. A *Craig interpolant* is a constraint I such that $A \models I$ and $B \models \neg I$, and such that $fv(I) \subseteq fv(A) \cap fv(B)$. The existence of an interpolant implies that $A \wedge B$ is unsatisfiable. We say that a constraint language has the *interpolation property* if also the opposite holds: whenever $A \wedge B$ is unsatisfiable, there is an interpolant I.

4.1 Horn Clauses

To define the concept of Horn clauses, we fix a set $\mathcal{R}$ of uninterpreted fixed-arity *relation symbols,* disjoint from $\mathcal{P}$ and $\mathcal{F}$. A *Horn clause* is a formula $C \wedge B_1 \wedge \cdots \wedge B_n \rightarrow H$ where

- C is a constraint over $\mathcal{F}, \mathcal{P}, \mathcal{X}$;
- each B_i is an application $p(t_1, \ldots, t_k)$ of a relation symbol $p \in \mathcal{R}$ to first-order terms over $\mathcal{F}, \mathcal{X}$;
- H is similarly either an application $p(t_1, \ldots, t_k)$ of $p \in \mathcal{R}$ 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 $C = \mathit{true}$, we usually leave out C and just write $B_1 \wedge \cdots \wedge B_n \rightarrow H$. First-order variables (from $\mathcal{X}$) in a clause are considered implicitly universally quantified; relation symbols represent set-theoretic relations over the universe U of a structure $(U, I) \in \mathcal{S}$. Notions like (un)satisfiability and entailment generalise straightforwardly to formulae with relation symbols.

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

$$\begin{aligned} \mathit{sol}(C \wedge p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n) \rightarrow p(\bar{t})) &= C \wedge \mathit{sol}(p_1)[\bar{t}_1] \wedge \cdots \wedge \mathit{sol}(p_n)[\bar{t}_n] \rightarrow \mathit{sol}(p)[\bar{t}] \\ \mathit{sol}(C \wedge p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n) \rightarrow \mathit{false}) &= C \wedge \mathit{sol}(p_1)[\bar{t}_1] \wedge \cdots \wedge \mathit{sol}(p_n)[\bar{t}_n] \rightarrow \mathit{false} \end{aligned}$$

Definition 1 (Solvability). *Let $\mathcal{HC}$ be a set of Horn clauses over relation symbols $\mathcal{R}$.*

1. *$\mathcal{HC}$ is called semantically solvable if for every structure $(U, I) \in \mathcal{S}$ there is an interpretation of the relation symbols $\mathcal{R}$ as set-theoretic relations over U such the universally quantified closure $Cl_\forall(h)$ of every clause $h \in \mathcal{HC}$ holds in (U, I).*
2. *A $\mathcal{HC}$ is called syntactically solvable if there is a relation symbol assignment sol such that for every structure $(U, I) \in \mathcal{S}$ and every clause $h \in \mathcal{HC}$ it is the case that $Cl_\forall(\mathit{sol}(h))$ is satisfied.*

Note that, in the special case when $\mathcal{S}$ contains only one structure, $\mathcal{S} = \{(U, I)\}$, semantic solvability reduces to the existence of relations interpreting $\mathcal{R}$ that extend the structure (U, I) in such a way to make all clauses true. In other words, Horn clauses are solvable in a structure if and only if the extension of the theory of (U, I) by relation symbols $\mathcal{R}$ in the vocabulary and by given Horn clauses as axioms is consistent.

A set $\mathcal{HC}$ of Horn clauses induces a *dependence relation* $\rightarrow_{\mathcal{HC}}$ on $\mathcal{R}$, defining $p \rightarrow_{\mathcal{HC}} q$ if there is a Horn clause in $\mathcal{HC}$ that contains p in its head, and q in the body. The set $\mathcal{HC}$ is called *recursion-free* if $\rightarrow_{\mathcal{HC}}$ is acyclic, and *recursive* otherwise. In the next sections we study the solvability problem for recursion-free Horn clauses and then show how to use such results in general Horn clause verification systems.

5 The Relationship between Craig Interpolation and Horn Clauses

It has become common to work with generalised forms of Craig interpolation, such as inductive sequences of interpolants, tree interpolants, and restricted DAG interpolants. We show that a variety of such interpolation approaches can be reduced to recursion-free Horn clauses. Recursion-free Horn clauses thus provide a general framework unifying and subsuming a number of earlier notions. As a side effect, we can formulate a general theorem about existence of the individual kinds of interpolants in Sect. 6, applicable to any constraint language with the (binary) interpolation property.

An overview of the relationship between specific forms of interpolation and specific fragments of recursions-free Horn clauses is given in Table 1, and will be explained in more detail in the rest of this section. Table 1 refers to the following fragments of recursion-free Horn clauses:

Form of interpolation	**Fragment of Horn clauses**
Binary interpolation [10, 24] $A \wedge B$	Pair of Horn clauses $A \rightarrow p(\bar{x}),\ B \wedge p(\bar{x}) \rightarrow \mathit{false}$ with $\{\bar{x}\} = \mathit{fv}(A) \cap \mathit{fv}(B)$
Inductive interpolant seq. [19, 26] $T_1 \wedge T_2 \wedge \cdots \wedge T_n$	Linear tree-like Horn clauses $T_1 \rightarrow p_1(\bar{x}_1),\ \ p_1(\bar{x}_1) \wedge T_2 \rightarrow p_2(\bar{x}_2),\ \ \ldots$ with $\{\bar{x}_i\} = \mathit{fv}(T_1, \ldots, T_i) \cap \mathit{fv}(T_{i+1}, \ldots, T_n)$
Tree interpolants [18, 23]	Tree-like Horn clauses
Restricted DAG interpolants [1]	Linear Horn clauses
Disjunctive interpolants [29]	Body disjoint Horn clauses

Table 1. Equivalence of interpolation problems and systems of Horn clauses.

Definition 2 (Horn clause fragments). *We say that a finite, recursion-free set $\mathcal{HC}$ of Horn clauses*

1. *is linear if the body of each Horn clause contains at most one relation symbol,*
2. *is body-disjoint if for each relation symbol p there is at most one clause containing p in its body; furthermore, every clause contains p at most once;*
3. *is head-disjoint if for each relation symbol p there is at most one clause containing p in its head;*
4. *is tree-like [17] if it is body-disjoint and head-disjoint.*

Theorem 1 (Interpolation and Horn clauses). *For each line of Table 1 it holds that:*

1. *an interpolation problem of the stated form can be polynomially reduced to (syntactically) solving a set of Horn clauses, in the stated fragment;*
2. *solving a set of Horn clauses (syntactically) in the stated fragment can be polynomially reduced to solving a sequence of interpolation problems of the stated form.*

5.1 Binary Craig Interpolants [10, 24]

The simplest form of Craig interpolation is the derivation of a constraint I such that $A \models I$ and $I \models \neg B$, and such that $\mathit{fv}(I) \subseteq \mathit{fv}(A) \cap \mathit{fv}(B)$. Such derivation is typically constructed by efficiently processing the proof of unsatisfiability of $A \wedge B$. To encode a binary interpolation problem into Horn clauses, we first determine the set $\bar{x} = \mathit{fv}(A) \cap \mathit{fv}(B)$ of variables that can possibly occur in the interpolant. We then pick a relation symbol p of arity $|\bar{x}|$, and define two Horn clauses expressing that $p(\bar{x})$ is an interpolant:

$$A \rightarrow p(\bar{x}), \qquad B \wedge p(\bar{x}) \rightarrow \mathit{false}$$

It is clear that every syntactic solution for the two Horn clauses corresponds to an interpolant of $A \wedge B$.

5.2 Inductive Sequences of Interpolants [19, 26]

Given an unsatisfiable conjunction $T_1 \wedge \ldots \wedge T_n$ (in practice, often corresponding to an infeasible path in a program), an *inductive sequence of interpolants* is a sequence $I_0, I_1, \ldots, I_n$ of formulae such that

1. $I_0 = \mathit{true}$, $I_n = \mathit{false}$,
2. for all $i \in \{1, \ldots, n\}$, the entailment $I_{i-1} \wedge T_i \models I_i$ holds, and
3. for all $i \in \{0, \ldots, n\}$, it is the case that $\mathit{fv}(I_i) \subseteq \mathit{fv}(T_1, \ldots, T_i) \cap \mathit{fv}(T_{i+1}, \ldots, T_n)$.

While inductive sequences can be computed by repeated computation of binary interpolants [19], more efficient solvers have been developed that derive a whole sequence of interpolants simultaneously [8, 9, 23].

Inductive sequences as linear tree-like Horn clauses. An inductive sequence of interpolants can straightforwardly be encoded as a set of linear Horn clauses, by introducing a fresh relation symbol p_i for each interpolant I_i to be computed. The arguments of the relation symbols have to be chosen reflecting condition 3 of the definition of interpolant sequences: for each $i \in \{0, \ldots, n\}$, we assume that $\bar{x}_i = \mathit{fv}(T_1, \ldots, T_i) \cap \mathit{fv}(T_{i+1}, \ldots, T_n)$ is the vector of variables that can occur in I_i. Conditions 1 and 2 are then represented by the following Horn clauses:

$$p_0(\bar{x}_0), \quad p_0(\bar{x}_0) \wedge T_1 \rightarrow p_1(\bar{x}_1), \quad p_1(\bar{x}_1) \wedge T_2 \rightarrow p_2(\bar{x}_2), \quad \ldots, \quad p_n(\bar{x}_n) \rightarrow \mathit{false}$$

Linear tree-like Horn clauses as inductive sequences. Suppose $\mathcal{HC}$ is a finite, recursion-free, linear, and tree-like set of Horn clauses. We can solve the system of Horn clauses by computing one inductive sequence of interpolants for every connected component of the $\rightarrow_{\mathcal{HC}}$-graph. First, each clause is normalised in a manner similar to [15]: for every relation symbol p, we fix a unique vector of variables $\bar{x}_p$, and rewrite $\mathcal{HC}$ such that p only occurs in the form $p(\bar{x}_p)$; this is possible since $\mathcal{HC}$ is recursion-free and body-disjoint. We then ensure, through renaming, that every variable x that is not argument of a relation symbol occurs in at most one clause. A connected component then represents Horn clauses

$$C_1 \rightarrow p_1(\bar{x}_1), \quad C_2 \wedge p_1(\bar{x}_1) \rightarrow p_2(\bar{x}_2), \quad C_3 \wedge p_2(\bar{x}_2) \rightarrow p_3(\bar{x}_3), \quad \ldots, \quad C_n \wedge p_n(\bar{x}_n) \rightarrow \mathit{false}\,.$$

(If the first or the last of the clauses is missing, we assume that its constraint is *false*.) Any inductive sequence of interpolants for $C_1 \wedge C_2 \wedge C_3 \wedge \cdots \wedge C_n$ solves the clauses.

5.3 Tree Interpolants [18, 23]

Tree interpolants strictly generalise inductive sequences of interpolants, and are designed with the application of inter-procedural verification in mind: in this context, the tree structure of the interpolation problem corresponds to (a part of) the call graph of a program. Tree interpolation problems correspond to recursion-free tree-like sets of Horn clauses.

Suppose (V, E) is a finite directed tree, writing $E(v, w)$ to express that the node w is a direct child of v. Further, suppose $\phi : V \rightarrow \mathit{Constr}$ is a function that labels each node v of the tree with a formula $\phi(v)$. A labelling function $I : V \rightarrow \mathit{Constr}$ is called a *tree interpolant* (for (V, E) and ϕ) if the following properties hold:

1. for the root node $v_0 \in V$, it is the case that $I(v_0) = \mathit{false}$,
2. for any node $v \in V$, the following entailment holds:

$$\phi(v) \wedge \bigwedge_{(v,w)\in E} I(w) \;\models\; I(v)\,,$$

3. for any node $v \in V$, every non-logical symbol (in our case: variable) in $I(v)$ occurs both in some formula $\phi(w)$ for w such that $E^*(v, w)$, and in some formula $\phi(w')$ for some w' such that $\neg E^*(v, w')$. (E^* is the reflexive transitive closure of E).

Since the case of tree interpolants is instructive for solving recursion-free sets of Horn clauses in general, we give a result about the existence of tree interpolants. The proof of the lemma computes tree interpolants by repeated derivation of binary interpolants; however, as for inductive sequences of interpolants, there are solvers that can compute all formulae of a tree interpolant simultaneously [16, 17, 23].

Lemma 1. *Suppose the constraint language Constr that has the interpolation property. Then a tree (V, E) with labelling function $\phi : V \to Constr$ has a tree interpolant I if and only if $\bigwedge_{v\in V} \phi(v)$ is unsatisfiable.*

Proof. "⇒" follows from the observation that every interpolant $I(v)$ is a consequence of the conjunction $\bigwedge_{(v,w)\in E^+} \phi(w)$.

"⇐": let $v_1, v_2, \ldots, v_n$ be an inverse topological ordering of the nodes in (V, E), i.e., an ordering such that $\forall i, j.\ (E(v_i, v_j) \Rightarrow i > j)$. We inductively construct a sequence of formulae $I_1, I_2, \ldots, I_n$, such that for every $i \in \{1, \ldots, n\}$ the following properties hold:

1. the following conjunction is unsatisfiable:

$$\bigwedge\{I_k \mid k \le i,\ \forall j.\ (E(v_j, v_k) \Rightarrow j > i)\} \;\wedge\; \Big(\phi(v_{i+1}) \wedge \phi(v_{i+2}) \wedge \cdots \wedge \phi(v_n)\Big) \qquad (1)$$

2. the following entailment holds:

$$\phi(v_i) \wedge \bigwedge_{(v_i,v_j)\in E} I_j \;\models\; I_i$$

3. every non-logical symbol in I_i occurs both in a formula $\phi(w)$ with $E^*(v_i, w)$, and in a formula $\phi(w')$ with $\neg E^*(v_i, w')$.

Assume that the formulae $I_1, I_2, \ldots, I_i$ have been constructed, for $i \in \{0, \ldots, n-1\}$. We then derive the next interpolant I_{i+1} by solving the binary interpolation problem

$$\Big(\phi(v_{i+1}) \wedge \bigwedge_{E(v_{i+1},v_j)} I_j\Big) \;\wedge\; \Big(\bigwedge\{I_k \mid k \le i,\ \forall j.\ (E(v_j, v_k) \Rightarrow j > i+1)\} \;\wedge\; \phi(v_{i+2}) \wedge \cdots \wedge \phi(v_n)\Big) \qquad (2)$$

That is, we construct I_{i+1} so that the following entailments hold:

$$\phi(v_{i+1}) \wedge \bigwedge_{E(v_{i+1},v_j)} I_j \;\models\; I_{i+1},$$

$$\bigwedge\{I_k \mid k \le i,\ \forall j.\ (E(v_j, v_k) \Rightarrow j > i+1)\} \;\wedge\; \phi(v_{i+2}) \wedge \cdots \wedge \phi(v_n) \;\models\; \neg I_{i+1}$$

Furthermore, I_{i+1} only contains non-logical symbols that are common to the left and the right side of the conjunction.

Note that (2) is equivalent to (1), therefore unsatisfiable, and a well-formed interpolation problem. It is also easy to see that the properties 1–3 hold for I_{i+1}. Also, we can easily verify that the labelling function $I : v_i \mapsto I_i$ is a solution for the tree interpolation problem defined by (V, E) and ϕ. □

Tree interpolation as tree-like Horn clauses. The encoding of a tree interpolation problem as a tree-like set of Horn clauses is very similar to the encoding for inductive sequences of interpolants. We introduce a fresh relation symbol p_v for each node $v \in V$ of a tree interpolation problem $(V, E), \phi$, assuming that for each $v \in V$ the vector $\bar{x}_v = \bigcup_{E^*(v,w)} fv(\phi(w)) \cap \bigcup_{\neg E^*(v,w)} fv(\phi(w))$ represents the set of variables that can occur in the interpolant $I(v)$. The interpolation problem is then represented by the following clauses:

$$p_0(\bar{x}_0) \rightarrow \mathit{false}, \quad \left\{ \phi(v) \wedge \bigwedge_{(v,w)\in E} p_w(\bar{x}_w) \rightarrow p_v(\bar{x}_v) \right\}_{v\in V}$$

Tree-like Horn clauses as tree interpolation. Suppose $\mathcal{HC}$ is a finite, recursion-free, and tree-like set of Horn clauses. We can solve the system of Horn clauses by computing a tree interpolant for every connected component of the $\rightarrow_{\mathcal{HC}}$-graph. As before, we first normalise the Horn clauses by fixing, for every relation symbol p, a unique vector of variables $\bar{x}_p$, and rewriting $\mathcal{HC}$ such that p only occurs in the form $p(\bar{x}_p)$. We also ensure that every variable x that is not argument of a relation symbol occurs in at most one clause. The tree interpolation graph (V, E) is then defined by choosing the set $V = \mathcal{R} \cup \{\mathit{false}\}$ of relation symbols as nodes, and the child relation $E(p, q)$ to hold whenever p occurs as head, and q within the body of a clause. The labelling function ϕ is defined by $\phi(p) = C$ whenever there is a clause with head symbol p and constraint C, and $\phi(p) = \mathit{false}$ if p does not occur as head of any clause.

Example 1. We consider a subset of the Horn clauses given in Fig. 2:

(1) r1(X, Res) ← **true**
(2) r2(X', Res) ← r1(X, Res) ∧ X' ≥ 0
(3) r3(X, Res') ← r2(X, Res) ∧ rf(X, Res')
(5) **false** ← r3(X, Res) ∧ Res ≠ X + 1
(6) r5(N, Rec, Tmp) ← **true**
(9) r8(N, Rec, Tmp) ← r5(N, Rec, Tmp) ∧ N ≤ 0
(11) r9(N, Rec', Tmp) ← r8(N, Rec, Tmp) ∧ Rec' = 1
(12) rf (N, Rec) ← r9(N, Rec, Tmp)

Note that this recursion-free subset of the clauses is body-disjoint and head-disjoint, and thus tree-like. Since the complete set of clauses in Fig. 2 is solvable, also any subset is; in order to compute a (syntactic) solution of the clauses, we set up the corresponding tree interpolation problem. Fig. 4 shows the tree with the labelling ϕ to be interpolated (in grey), as well as the head literals of the clauses generating the nodes of the tree. A tree interpolant solving the interpolation problem is given in Fig. 5. The tree interpolant can straightforwardly be mapped to a solution of the original tree-like Horn, for instance we set $r_8(n_8, rec_8, tmp_8) = (n_8 \leq 0)$ and $r_9(n_9, rec_9, tmp_9) = (n_9 \leq -1 \vee (rec_9 = 1 \wedge n_9 = 0))$.

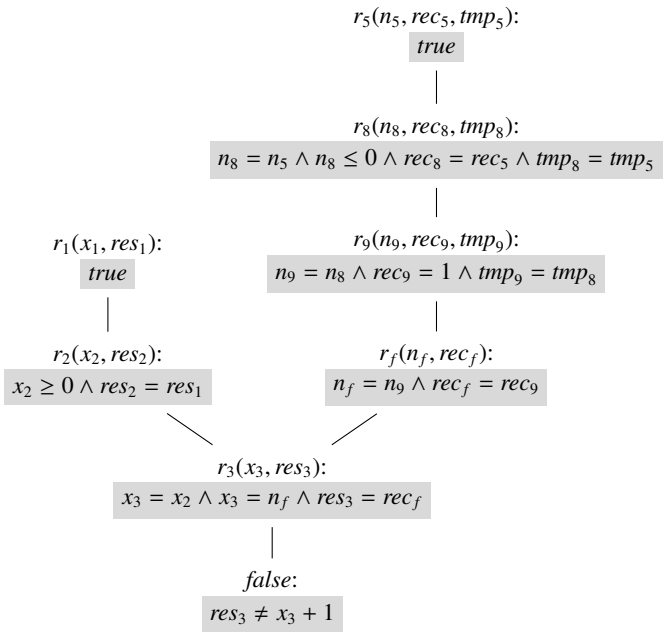


Fig. 4. Tree interpolation problem for the clauses in Example 1

Symmetric Interpolants A special case of tree interpolants, *symmetric interpolants,* was introduced in [25]. Symmetric interpolants are equivalent to tree interpolants with a flat tree structure (V, E), i.e., $V = \{root, v_1, \ldots, v_n\}$, where the nodes $v_1, \ldots, v_n$ are the direct children of *root*.

5.4 Restricted (and Unrestricted) DAG Interpolants [1]

Restricted DAG interpolants are a further generalisation of inductive sequence of interpolants, introduced for the purpose of reasoning about multiple paths in a program simultaneously [1]. Suppose (V, E, en, ex) is a finite connected DAG with entry node $en \in V$ and exit node $ex \in V$, further $\mathcal{L}_E : E \to Constr$ a labelling of edges with constraints, and $\mathcal{L}_V : V \to Constr$ a labelling of vertices. A *restricted DAG interpolant* is a mapping $I : V \to Constr$ with

1. $I(en) = true$, $I(ex) = false$,
2. for all $(v, w) \in E$ the entailment $I(v) \wedge \mathcal{L}_V(v) \wedge \mathcal{L}_E(v, w) \models I(w) \wedge \mathcal{L}_V(w)$ holds, and
3. for all $v \in V$ it is the case that[4]

$$fv(I(v)) \subseteq \Big(\bigcup_{(a,v)\in E} fv(\mathcal{L}_E(a, v)) \Big) \cap \Big(\bigcup_{(v,a)\in E} fv(\mathcal{L}_E(v, a)) \Big) .$$

[4] The definition of DAG interpolants in [1, Def. 4] implies that $fv(I(v)) = \emptyset$ for every interpolant $I(v), v \in V$, i.e., only trivial interpolants are allowed. We assume that this is a mistake in [1, Def. 4], and corrected the definition as shown here.

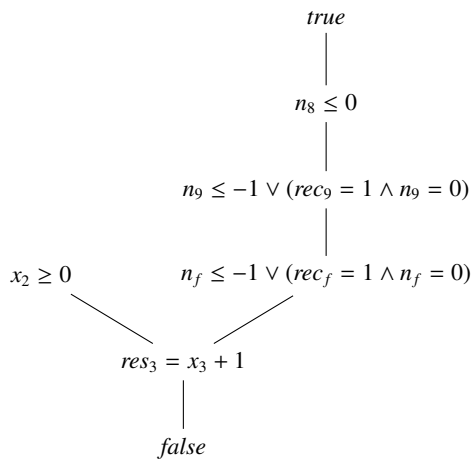


Fig. 5. Tree interpolant solving the interpolation problem in Fig. 4

The UFO verification system [3] is able to compute DAG interpolants, based on the interpolation functionality of MathSAT [9]. We can observe that DAG interpolants (despite their name) are incomparable in expressiveness to tree interpolation. This is because DAG interpolants correspond to *linear* Horn clauses, and might have shared relation symbol in bodies, while tree interpolants correspond to *possibly nonlinear tree-like* Horn clauses, but do not allow shared relation symbols in bodies. Nevertheless, it is possible to reduce DAG interpolants to tree interpolants, but only at the cost of a potentially exponential growth in the number of clauses.

Encoding of restricted DAG interpolants as linear Horn clauses. For every $v \in V$, let

$$\{\bar{x}_v\} \;=\; \Big(\bigcup_{(a,v)\in E} fv(\mathcal{L}_E(a,v))\Big) \cap \Big(\bigcup_{(v,a)\in E} fv(\mathcal{L}_E(v,a))\Big)$$

be the variables allowed in the interpolant to be computed for v, and p_v be a fresh relation symbol of arity $|\bar{x}_v|$. The interpolation problem is then defined by the following set of linear Horn clauses:

$$\begin{array}{rl}
\text{For each } (v,w) \in E\text{:} & \mathcal{L}_V(v) \wedge \mathcal{L}_E(v,w) \wedge p_v(\bar{x}_v) \rightarrow p_w(\bar{x}_w), \\
& \mathcal{L}_V(v) \wedge \neg\mathcal{L}_V(w) \wedge \mathcal{L}_E(v,w) \wedge p_v(\bar{x}_v) \rightarrow \mathit{false}, \\
\text{For } en, ex \in V\text{:} & \mathit{true} \rightarrow p_{en}(\bar{x}_{en}), \qquad p_{ex}(\bar{x}_{ex}) \rightarrow \mathit{false}
\end{array}$$

Encoding of linear Horn clauses as DAG interpolants. Suppose $\mathcal{HC}$ is a finite, recursion-free, and linear set of Horn clauses. We can solve the system of Horn clauses by computing a DAG interpolant for every connected component of the $\rightarrow_{\mathcal{HC}}$-graph. As in Sect. 5.2, we normalise Horn clauses by fixing a unique vector $\bar{x}_p$ of argument variables for each relation symbol p, and ensure that every non-argument variable x occurs

in at most one clause. We also assume that multiple clauses $C \wedge p(\bar{x}_p) \rightarrow q(\bar{x}_q)$ and $D \wedge p(\bar{x}_p) \rightarrow q(\bar{x}_q)$ with the same relation symbols are merged to $(C \vee D) \wedge p(\bar{x}_p) \rightarrow q(\bar{x}_q)$.

Let $\{p_1, \ldots, p_n\}$ be all relation symbols of one connected component. We then define the DAG interpolation problem $(V, E, en, ex), \mathcal{L}_E, \mathcal{L}_V$ by

- the vertices $V = \{p_1, \ldots, p_n\} \cup \{en, ex\}$, including two fresh nodes en, ex,
- the edge relation

$$\begin{aligned} E \;=\; & \{(p,q) \mid \text{there is a clause } C \wedge p(\bar{x}_p) \rightarrow q(\bar{x}_q) \in \mathcal{HC}\} \\ & \cup \{(en,p) \mid \text{there is a clause } D \rightarrow p(\bar{x}_p) \in \mathcal{HC}\} \\ & \cup \{(p,ex) \mid \text{there is a clause } E \wedge p(\bar{x}_p) \rightarrow \mathit{false} \in \mathcal{HC}\}\,, \end{aligned}$$

- for each $(v, w) \in E$, the edge labelling

$$\mathcal{L}_E(v,w) \;=\; \begin{cases} C \wedge \bar{x}_v = \bar{x}_v \wedge \bar{x}_w = \bar{x}_w & \text{if } C \wedge v(\bar{x}_v) \rightarrow w(\bar{x}_w) \in \mathcal{HC} \\ D \wedge \bar{x}_w = \bar{x}_w & \text{if } v = en \text{ and } D \rightarrow w(\bar{x}_w) \in \mathcal{HC} \\ E \wedge \bar{x}_v = \bar{x}_v & \text{if } w = ex \text{ and } E \wedge v(\bar{x}_v) \rightarrow \mathit{false} \in \mathcal{HC} \end{cases}$$

  Note that the labels include equations like $\bar{x}_v = \bar{x}_v$ to ensure that the right variables are allowed to occur in interpolants.
- for each $v \in V$, the node labelling $\mathcal{L}_V(v) = \mathit{true}$.

By checking the definition of DAG interpolants, it can be verified that every interpolant solving the problem $(V, E, en, ex), \mathcal{L}_E, \mathcal{L}_V$ is also a solution of the linear Horn clauses.

5.5 Disjunctive Interpolants [29]

Disjunctive interpolants were introduced in [29] as a generalisation of tree interpolants. Disjunctive interpolants resemble tree interpolants in the sense that the relationship of the components of an interpolant is defined by a tree; in contrast to tree interpolants, however, this tree is an and/or-tree: branching in the tree can represent either *conjunctions* or *disjunctions*. Disjunctive interpolants correspond to sets of body-disjoint Horn clauses; in this representation, and-branching is encoded by clauses with multiple body literals (like with tree interpolants), while or-branching is interpreted as multiple clauses sharing the same head symbol. For a detailed account on disjunctive interpolants, we refer the reader to [29].

The solution of body-disjoint Horn clauses can be computed by solving a sequence of tree-like sets of Horn clauses:

Lemma 2. *Let $\mathcal{HC}$ be a finite set of recursion-free body-disjoint Horn clauses. $\mathcal{HC}$ has a syntactic/semantic solution if and only if every maximum tree-like subset of $\mathcal{HC}$ has a syntactic/semantic solution.*

Proof. We outline direction "$\Leftarrow$" for syntactic solutions. Solving the tree-like subsets of $\mathcal{HC}$ yields, for each relation symbol $p \in \mathcal{R}$, a set SC_p of solution constraints. A global solution of $\mathcal{HC}$ can be constructed by forming a positive Boolean combination of the constraints in SC_p for each $p \in \mathcal{R}$. □

Example 2. We consider a recursion-free unwinding of the Horn clauses in Fig. 2. To make the set of clauses body-disjoint, the clause (6), (9), (11), (12) were duplicated, introducing primed copies of all relation symbols involved. The clauses are not head-disjoint, since (10) and (11) share the same head symbol:

(1) r1(X, Res) ← **true**
(2) r2(X', Res) ← r1(X, Res) ∧ X' ≥ 0
(3) r3(X, Res') ← r2(X, Res) ∧ rf(X, Res')
(5) **false** ← r3(X, Res) ∧ Res ≠ X + 1

(6) r5(N, Rec, Tmp) ← **true**
(7) r6(N, Rec, Tmp) ← r5(N, Rec, Tmp) ∧ N > 0
(8) r7(N, Rec, Tmp') ← r6(N, Rec, Tmp) ∧ rf'(N − 1, Tmp')
(9) r8(N, Rec, Tmp) ← r5(N, Rec, Tmp) ∧ N ≤ 0
(10) r9(N, Rec', Tmp) ← r7(N, Rec, Tmp) ∧ Rec' = Tmp + 1
(11) r9(N, Rec', Tmp) ← r8(N, Rec, Tmp) ∧ Rec' = 1
(12) rf (N, Rec) ← r9(N, Rec, Tmp)

(6') r5'(N, Rec, Tmp) ← **true**
(9') r8'(N, Rec, Tmp) ← r5'(N, Rec, Tmp) ∧ N ≤ 0
(11') r9'(N, Rec', Tmp) ← r8'(N, Rec, Tmp) ∧ Rec' = 1
(12') rf '(N, Rec) ← r9'(N, Rec, Tmp)

There are two maximum tree-like subsets: $T_1 = \{(1),(2),(3),(5),(6),(9),(11),(12)\}$, and $T_2 = \{(1),(2),(3),(5),(6),(7),(8),(10),(12),(6'),(9'),(11'),(12')\}$. The subset T_1 has been discussed in Example 1. In the same way, it is possible to construct a solution for T_2 by solving a tree interpolation problem. The two solutions can be combined to construct a solution of $T_1 \cup T_2$:

	T_1	T_2	$T_1 \cup T_2$
$r_1(x,r)$	*true*	*true*	*true*
$r_2(x,r)$	$x \geq 0$	*true*	$x \geq 0$
$r_3(x,r)$	$r = x+1$	$r = x+1$	$r = x+1$
$r_5(n,c,t)$	*true*	*true*	*true*
$r_6(n,c,t)$	–	$n \geq 1$	$n \geq 1$
$r_7(n,c,t)$	–	$t = n$	$t = n$
$r_8(n,c,t)$	$n \leq 0$	–	$n \leq 0$
$r_9(n,c,t)$	$n \leq -1 \vee (c = 1 \wedge n = 0)$	$c = n+1$	$n \leq -1 \vee c = n+1$
$r_f(n,c)$	$n \leq -1 \vee (c = 1 \wedge n = 0)$	$c = n+1$	$n \leq -1 \vee c = n+1$
$r'_5(n,c,t)$	–	*true*	*true*
$r'_8(n,c,t)$	–	$n \leq 0$	$n \leq 0$
$r'_9(n,c,t)$	–	$n \leq -1 \vee (c = 1 \wedge n = 0)$	$n \leq -1 \vee (c = 1 \wedge n = 0)$
$r'_f(n,c,t)$	–	$n \leq -1 \vee (c = 1 \wedge n = 0)$	$n \leq -1 \vee (c = 1 \wedge n = 0)$

In particular, the disjunction of the two interpretations of $r_9(n,c,t)$ has to be used, in order to satisfy both (10) and (11) (similarly for $r_f(n,c)$). In contrast, the conjunction of the interpretations of $r_2(n,c,t)$ is needed to satisfy (3).

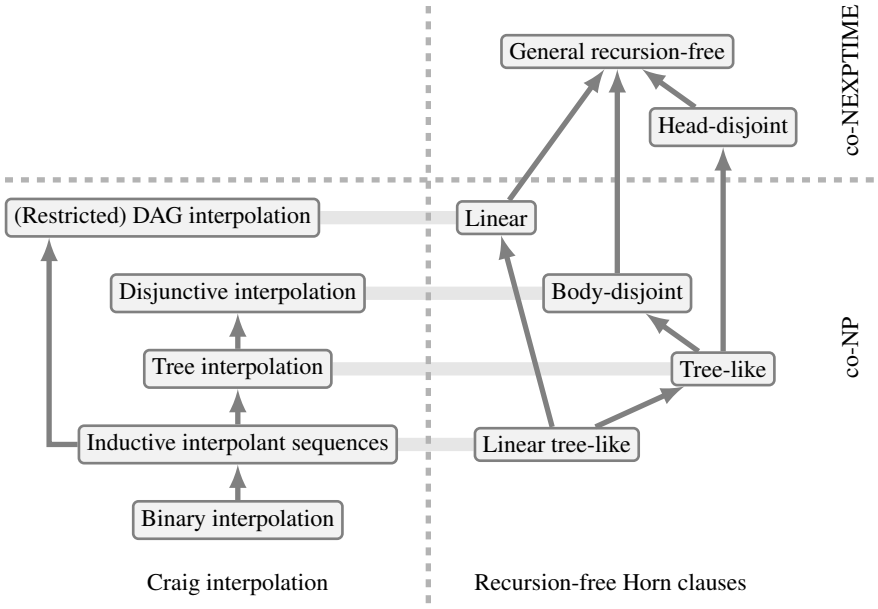


Fig. 6. Relationship between different forms of Craig interpolation, and different fragments of recursion-free Horn clauses. An arrow from A to B expresses that problem A is (strictly) subsumed by B. The complexity classes "co-NP" and "co-NEXPTIME" refer to the problem of checking solvability of Horn clauses over quantifier-free Presburger arithmetic.

6 The Complexity of Recursion-free Horn Clauses

We give an overview of the considered fragments of recursion-free Horn clauses, and the corresponding interpolation problem, in Fig. 6. The diagram also shows the complexity of deciding (semantic or syntactic) solvability of a set of Horn clauses, for Horn clauses over the constraint language of quantifier-free Presburger arithmetic. Most of the complexity results occur in [29], but in addition we use the following two observations:

Lemma 3. *Semantic solvability of recursion-free linear Horn clauses over the constraint language of quantifier-free Presburger arithmetic is in co-NP.*

Proof. A set $\mathcal{HC}$ of recursion-free linear Horn clauses is solvable if and only if the expansion $exp(\mathcal{HC})$ is unsatisfiable [29]. For linear clauses, $exp(\mathcal{HC})$ is a disjunction of (possibly) exponentially many formulae, each of which is linear in the size of $exp(\mathcal{HC})$. Consequently, satisfiability of $exp(\mathcal{HC})$ is in NP, and unsatisfiability in co-NP. □

Lemma 4. *Semantic solvability of recursion-free head-disjoint Horn clauses over the constraint language of quantifier-free Presburger arithmetic is co-NEXPTIME-hard.*

Proof. The proof given in [29] for co-NEXPTIME-hardness of recursion-free Horn clauses over quantifier-free Presburger arithmetic can be adapted to only require head-

disjoint clauses. This is because a single execution step of a non-deterministic Turing machine can be expressed as quantifier-free Presburger formula. □

7 Towards a Library of Interpolation Benchmarks

In order to support the development of interpolation engines, Horn solvers, and verification systems, we have started to collect relevant benchmarks of recursion-free Horn clauses, categorised according to the classes determined in the previous sections.[5] The benchmarks have been extracted from runs of the model checker Eldarica [29], which processes systems of (usually recursive) Horn clauses by iteratively solving recursion-free unwindings, as outlined in Sect. 3. For each recursive verification problem, in this way a set of recursion-free systems of Horn clauses (of varying size) can be synthesised. The benchmarks can be used to evaluate both Horn solvers and interpolation engines, according to the correspondence in Fig. 6.

At the moment, our benchmarks are extracted from the verification problems in [29], and formulated over the constraint language of linear integer arithmetic; in the future, it is planned to also include other constraint languages, including rational arithmetic and the theory of arrays. The benchmarks are stored in SMT-LIB 2 format [5]. All of the benchmarks can be solved by Eldarica, and by the Horn solving engine in Z3 [20].

The current number of available benchmarks is provided in the table below. In order to evaluate the effectiveness of ordinary SMT quantifier handling on the Horn benchmarks, we also ran Z3 [11] (without the Horn engine) on the benchmarks. The results show that two engines that have knowledge of Horn clauses, Eldarica and Z3-Horn, solve all of the benchmarks (with Z3's well-engineered engine faster than Eldarica). In contrast, when Z3 is used without Horn extension, as a prover for quantified formulas, the default quantifier instantiation strategy proves to be too weak to solve all benchmarks.

Class	**#Benchmarks**	**Average Time**		**%Solved Z3**
		Eldarica	Z3	(without Horn)
General recursion-free	541	0.6	0.1	80%
Head-disjoint	991	0.1	0.1	85%
Linear	971	0.7	0.1	32%
Linear tree-like	1993	0.4	0.1	55%

Fig. 7. Average time in seconds of solving each category by Eldarica and Z3. The last column shows the percentage handled by Z3 without the Horn engine. Time-out was set to 60 seconds.

8 From Recursion-free Horn Clauses to Well-founded Clauses

It is natural to ask whether the considerations of the last sections also apply to clauses that are not Horn clauses (i.e., clauses that can contain multiple positive literals), pro-

[5] http://lara.epfl.ch/w/horn-nonrec-benchmarks

vided the clauses are "recursion-free." Is it possible, like for Horn clauses, to compute solutions of recursion-free clauses in general by means of computing Craig interpolants?

To investigate the situation for clauses that are not Horn, we first have to generalise the concept of clauses being recursion-free: the definition provided in Sect. 4, formulated with the help of the dependence relation $\rightarrow_{\mathcal{HC}}$, only applies to Horn clauses. For non-Horn clauses, we instead choose to reason about the absence of infinite propositional resolution derivations. Because the proposed algorithms [29] for solving recursion-free sets of Horn clauses all make use of *exhaustive expansion* or *inlining,* i.e., the construction of all derivations for a given set of clauses, the requirement that no infinite derivations exist is fundamental.[6]

Somewhat surprisingly, we observe that all sets of clauses without infinite derivations have the shape of Horn clauses, up to renaming of relation symbols. This means that procedures handling Horn clauses cover all situations in which we can hope to compute solutions with the help of Craig interpolation.

Since constraints and relation symbol arguments are irrelevant for this observation, the following results are entirely formulated on the level of propositional logic:

- a propositional *literal* is either a Boolean variable p, q, r (positive literals), or the negation $\neg p, \neg q, \neg r$ of a Boolean variable (negative literals).
- a propositional *clause* is a disjunction $p \vee \neg q \vee p$ of literals. The multiplicity of a literal is important, i.e., clauses could alternatively be represented as multi-sets of literals.
- a *Horn clause* is a clause that contains at most one positive literal.
- given a set $\mathcal{HC}$ of Horn clauses, we define the dependence relation $\rightarrow_{\mathcal{HC}}$ on Boolean variables by setting $p \rightarrow_{\mathcal{HC}} q$ if and only if there is a clause in $\mathcal{HC}$ in which p occurs positively, and q negatively (like in Sect. 4). The set $\mathcal{HC}$ is called *recursion-free* if $\rightarrow_{\mathcal{HC}}$ is acyclic.

We can now generalise the notion of a set of clauses being "recursion-free" to non-Horn clauses:

Definition 3. *A set C of propositional clauses has the termination property if no infinite sequence $c_0, c_1, c_2, c_3, \ldots$ of clauses exists, such that*

- *$c_0 \in C$ is an input clause, and*
- *for each $i \geq 1$, the clause c_i is derived by means of binary resolution from c_{i-1} and an input clause, using the rule*

$$\frac{C \vee p \qquad D \vee \neg p}{C \vee D}\,.$$

[6] We do not take subsumption between clauses, or loops in derivations into account. This means that a set of clauses might give rise to infinite derivations even if the set of derived clauses is finite. It is conceivable that notions of subsumption, or more generally the application of terminating saturation strategies [13], can be used to identify more general fragments of clauses for which syntactic solutions can effectively be computed. This line of research is future work.

Lemma 5. *A finite set $\mathcal{HC}$ of Horn clauses has the termination property if and only if it is recursion-free.*

Proof. "$\Leftarrow$" The acyclic dependence relation $\rightarrow_{\mathcal{HC}}$ induces a strict well-founded order $<$ on Boolean variables: $q \rightarrow_{\mathcal{HC}} p$ implies $p < q$. The order $<$ induces a well-founded order $\ll$ on Horn clauses:

$$\begin{aligned}(p \vee C) \ll (q \vee D) &\Leftrightarrow p > q \text{ or } (p = q \text{ and } C <_{ms} D)\\ C \ll (q \vee D) &\Leftrightarrow \mathit{true}\\ C \ll D &\Leftrightarrow C <_{ms} D\end{aligned}$$

where C, D only contain negative literals, and $<_{ms}$ is the (well-founded) multi-set extension of $<$ [12].

It is easy to see that a clause $C \vee D$ derived from two Horn clauses $C \vee p$ and $D \vee \neg p$ using the resolution rule is again Horn, and $(C \vee D) \ll (C \vee p)$ and $(C \vee D) \ll (D \vee \neg p)$. The well-foundedness of $\ll$ implies that any sequence of clauses as in Def. 3 is finite.

"$\Rightarrow$" If the dependence relation $\rightarrow_{\mathcal{HC}}$ has a cycle, we can directly construct a non-terminating sequence $c_0, c_1, c_2, \ldots$ of clauses. □

Definition 4 (Renamable-Horn [22]). *If A is a set of Boolean variables, and C is a set of clauses, then $r_A(C)$ is the result of replacing in C every literal whose Boolean variable is in A with its complement. C is called renamable-Horn if there is some set A of Boolean variables such that $r_A(C)$ is Horn.*

Theorem 2. *If a finite set C of clauses has the termination property, then it is renamable-Horn.*

Proof. Suppose C is formulated over the (finite) set $p_1, p_2, \ldots, p_n$ of Boolean variables. We construct a graph (V, E), with $V = \{p_1, p_2, \ldots, p_n, \neg p_1, \neg p_2, \ldots, \neg p_n\}$ being the set of all possible literals, and $(l, l') \in E$ if and only if there is a clause $\neg l \vee l' \vee C \in C$ (that means, a clause containing the literal l', and the literal l with reversed sign).[7]

The graph (V, E) is acyclic. To see this, suppose there is a cycle $l_1, l_2, \ldots, l_m, l_{m+1} = l_1$ in (V, E). Then there are clauses $c_1, c_2, \ldots, c_m \in C$ such that each c_i contains the literals $\neg l_i$ and l_{i+1}. We can then construct an infinite sequence $c_1 = d_0, d_1, d_2, \ldots$ of clauses, where each d_i (for $i > 1$) is obtained by resolving d_{i-1} with $c_{(i \bmod m)+1}$, contradicting the assumption that C has the termination property.

Since (V, E) is acyclic, there is a strict total order $<$ on V that is consistent with E, i.e., $(l, l') \in E$ implies $l < l'$.

Claim: if $p < \neg p$ for every Boolean variable $p \in \{p_1, p_2, \ldots, p_n\}$, then C is Horn.

Proof of the claim: suppose a non-Horn clause $p_i \vee p_j \vee C \in C$ exists (with $i \neq j$). Then $(\neg p_i, p_j) \in E$ and $(\neg p_j, p_i) \in E$, and therefore $\neg p_i < p_j$ and $\neg p_j < p_i$. Then also $\neg p_i < p_i$ or $\neg p_j < p_j$, contradicting the assumption that $p < \neg p$ for every Boolean variable p.

In general, choose $A = \{p_i \mid i \in \{1, \ldots, n\}, \neg p_i < p_i\}$, and consider the set $r_A(C)$ of clauses. The set $r_A(C)$ is Horn, since changing the sign of a Boolean variable $p \in A$

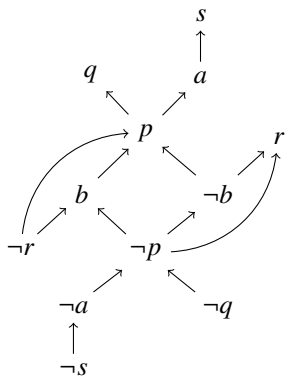
[7] This graph could equivalently be defined as the implication graph of the 2-sat problem introduced in [22], as a way of characterising whether a set of clauses is Horn.

has the effect of swapping the nodes $p, \neg p$ in the graph (V, E). Therefore, the new graph (V, E') has to be compatible with a strict total order $<$ such that $p < \neg p$ for every Boolean variable p, satisfying the assumption of the claim above. □

Example 3. We consider the following set of clauses:

$$C = \{\neg a \vee s,\ a \vee \neg p,\ p \vee \neg b,\ b \vee p \vee r,\ \neg p \vee q\}$$

By constructing all possible derivations, it can be shown that the set has the termination property. The graph (V, E), as constructed in the proof, is:



A strict total order that is compatible with the graph is:

$$\neg s < \neg q < \neg r < \neg a < \neg p < b < \neg b < r < p < q < a < s$$

From the order we can read off that we need to rename the variables $A = \{s, q, r, a, p\}$ in order to obtain a set of Horn clauses:

$$r_A(C) = \{a \vee \neg s,\ \neg a \vee p,\ \neg p \vee \neg b,\ b \vee \neg p \vee \neg r,\ p \vee \neg q\}$$

References

1. A. Albarghouthi, A. Gurfinkel, and M. Chechik. Craig interpretation. In *SAS*, 2012.
2. A. Albarghouthi, A. Gurfinkel, and M. Chechik. Whale: An interpolation-based algorithm for inter-procedural verification. In *VMCAI*, pages 39–55, 2012.
3. A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik. Ufo: A framework for abstraction- and interpolation-based software verification. In *CAV*, pages 672–678, 2012.
4. T. Ball, A. Podelski, and S. K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *TACAS'02*, volume 2280 of *LNCS*, page 158, 2002.
5. C. Barrett, A. Stump, and C. Tinelli. C.: The smt-lib standard: Version 2.0. Technical report, 2010.
6. N. Bjørner, K. McMillan, and A. Rybalchenko. Program verification as satisfiability modulo theories. In *SMT Workshop at IJCAR*, 2012.
7. M. P. Bonacina and M. Johansson. On interpolation in automated theorem proving. *(submitted)*, 2012.

8. A. Brillout, D. Kroening, P. Rümmer, and T. Wahl. An interpolating sequent calculus for quantifier-free Presburger arithmetic. *Journal of Automated Reasoning*, 47:341–367, 2011.
9. A. Cimatti, A. Griggio, and R. Sebastiani. Efficient generation of Craig interpolants in satisfiability modulo theories. *ACM Trans. Comput. Log.*, 12(1):7, 2010.
10. W. Craig. Linear reasoning. A new form of the Herbrand-Gentzen theorem. *The Journal of Symbolic Logic*, 22(3):250–268, September 1957.
11. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340. Springer-Verlag, 2008.
12. N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Commun. ACM*, 22(8):465–476, 1979.
13. C. Fermüller, A. Leitsch, U. Hustadt, and T. Tammet. Resolution decision procedures. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 25, pages 1791–1850. Elsevier, 2001.
14. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, 1997.
15. S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, 2012.
16. A. Gupta, C. Popeea, and A. Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In *POPL*, 2011.
17. A. Gupta, C. Popeea, and A. Rybalchenko. Solving recursion-free Horn clauses over LI+UIF. In *APLAS*, pages 188–203, 2011.
18. M. Heizmann, J. Hoenicke, and A. Podelski. Nested interpolants. In *POPL*, 2010.
19. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244. ACM, 2004.
20. K. Hoder and N. Bjørner. Generalized property directed reachability. In *SAT*, pages 157–171, 2012.
21. A. Lal, S. Qadeer, and S. K. Lahiri. Corral: A solver for reachability modulo theories. In *CAV*, 2012.
22. H. R. Lewis. Renaming a set of clauses as a Horn set. *J. ACM*, 25(1):134–135, Jan. 1978.
23. K. L. McMillan. iZ3 documentation. http://research.microsoft.com/en-us/um/redmond/projects/z3/iz3documentation.html.
24. K. L. McMillan. Interpolation and SAT-based model checking. In *CAV*, 2003.
25. K. L. McMillan. Applications of craig interpolation to model checking. In J. Marcinkowski and A. Tarlecki, editors, *CSL*, volume 3210 of *Lecture Notes in Computer Science*, pages 22–23. Springer, 2004.
26. K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, 2006.
27. K. L. McMillan and A. Rybalchenko. Solving constrained Horn clauses using interpolation. Technical Report MSR-TR-2013-6, Jan. 2013. http://research.microsoft.com/apps/pubs/default.aspx?id=180055.
28. 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.
29. P. Rümmer, H. Hojjat, and V. Kuncak. Disjunctive Interpolants for Horn-Clause Verification (Extended Technical Report). *ArXiv e-prints*, Jan. 2013. http://arxiv.org/abs/1301.4973.
30. O. Sery, G. Fedyukovich, and N. Sharygina. Interpolation-based function summaries in bounded model checking. In *Haifa Verification Conference (HVC)*, Haifa, 2011. Springer.
31. P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *Static Analysis Symposium (SAS)*, 2011.
32. M. Taghdiri and D. Jackson. Inferring specifications to detect errors in code. *Autom. Softw. Eng.*, 14(1):87–121, 2007.