# Generalised Interpolation by Solving Recursion-Free Horn Clauses

Ashutosh Gupta
IST, Austria
agupta@ist.ac.at

Corneliu Popeea
Technische Universität München
popeea@model.in.tum.de

Andrey Rybalchenko
Technische Universität München
Microsoft Research Cambridge
rybal@microsoft.com

In this paper we present INTERHORN, a solver for recursion-free Horn clauses. The main application domain of INTERHORN lies in solving interpolation problems arising in software verification. We show how a range of interpolation problems, including path, transition, nested, state/transition and well-founded interpolation can be handled directly by INTERHORN. By detailing these interpolation problems and their Horn clause representations, we hope to encourage the emergence of a common back-end interpolation interface useful for diverse verification tools.

## 1 Introduction

Interpolation is a key ingredient of a wide range of software verification tools that is used to compute approximations of sets and relations over program states, see e.g. [1, 2, 8, 11, 14, 17, 19, 20, 22, 24]. These approximations come in different forms, e.g., as path interpolation [15], transition interpolation [18], nested interpolation [14], state/transition interpolation [1], or well-founded interpolation [6]. As a result algorithms and tools for solving interpolation problems have become an important area of research contributing to the advances in state-of-the-art of software verification.

In this paper we present INTERHORN, a solver for constraints in form of recursion-free Horn clauses that can be applied on various interpolation problems occurring in software verification. INTERHORN takes as input clauses whose literals are either assertions in the theory of linear arithmetic or unknown relations. In addition, INTERHORN also accepts well-foundedness conditions on the unknown relations. The set of input clauses can represent either a DAG or a tree of dependencies between interpolants to be discovered. The output of INTERHORN is either an interpretation of unknown relations in terms of linear arithmetic assertions that turns the input clauses into valid implications over rationals/reals and satisfies well-foundedness conditions, or the statement that no such interpretation exists. INTERHORN is sound and complete for clauses without well-foundedness conditions. (INTERHORN is incomplete when well-foundedness conditions are present, since it relies on synthesis of linear ranking functions.) INTERHORN is a part of a general solver for recursive Horn clauses [8] and has already demonstrated its practicability in a software verification competition [7]. The main novelty offered by INTERHORN wrt. existing interpolating procedures [3–5, 9] lies in the ability to declaratively specify the interpolation problem as a set of recursion-free Horn clauses and the support for well-foundedness conditions.

## 2 Interpolation by solving recursion-free Horn clauses

In this section we provide examples of how interpolation related problems arising in software verification can be formulated as solving of recursion-free Horn clauses. This collection of examples is not exhaustive

Nikolaj Bjørner, Fabio Fioravanti, Andrey Rybalchenko,
Valerio Senni (Eds.): First Workshop on
Horn Clauses for Verification and Synthesis (HCVS 2014)
EPTCS 169, 2014, pp. 31–38, doi:10.4204/EPTCS.169.5

and serves as an illustration of the approach. We omit any description of how interpolation is used by verification methods, since it is out of scope of this paper, and rather focus on the form of interpolation problems and their representation as recursion-free Horn clauses. Further examples can be found in the literature, e.g., [8], as well as are likely to emerge in the future.

**Path interpolation** Interpolation can be used for the approximation of sets of states reachable by a program along a given path, see e.g. [15]. A flat program (transition system) consists of program variables $v$, an initiation condition $\mathit{init}(v)$, a set of program transitions $\{\mathit{next}_1(v,v'),\ldots,\mathit{next}_N(v,v')\}$, and a description of safe states $\mathit{safe}(v)$. A path is a sequence of program transitions.

Given a path $\mathit{next}_1(v,v'),\ldots,\mathit{next}_n(v,v')$, the path interpolation problem is to find assertions $I_0(v),I_1(v),\ldots,I_n(v)$ such that:

$$\begin{aligned}
&\mathit{init}(v) \rightarrow I_0(v),\\
&I_{k-1}(v) \wedge \mathit{next}_k(v,v') \rightarrow I_k(v'), \qquad \text{for each } k \in 1..n\\
&I_n(v) \rightarrow \mathit{safe}(v).
\end{aligned}$$

We observe that there are no recursive dependencies induced by the above implications between the interpolants to be discovered, i.e., $I_0(v)$ does not depend on any other interpolant, while $I_1(v)$ depends on $I_0(v)$, and $I_n(v)$ depends on $I_0(v),\ldots,I_{n-1}(v)$. InterHorn leverages such absence of dependency cycles in our solving algorithm, see Section 3.

**Transition interpolation** Interpolation can be applied to compute over-approximation of program transitions, see e.g. [18]. Given a path $\mathit{next}_1(v,v'),\ldots,\mathit{next}_n(v,v')$, a transition interpolation problem is to find $T_1(v,v'),\ldots,T_n(v,v')$ such that:

$$\begin{aligned}
&\mathit{next}_k(v,v') \rightarrow T_k(v,v'), && \text{for each } k \in 1..n\\
&\mathit{init}(v_0) \wedge T_1(v_0,v_1) \wedge \cdots \wedge T_n(v_{n-1},v_n) \rightarrow \mathit{safe}(v_n).
\end{aligned}$$

Again, we note there are no recursive dependencies between the assertions to be computed.

**Well-founded interpolation** We can also use interpolation in combination with additional well-foundedness constraints when proving program termination, see e.g. [6]. We assume a path $\mathit{stem}_1(v,v'),\ldots,\mathit{stem}_m(v,v')$ that contains transitions leading to a loop entry point, and a path $\mathit{loop}_1(v,v'),\ldots,\mathit{loop}_n(v,v')$ around the loop. A well-founded interpolation problem amounts to finding $I_0(v),I_1(v),\ldots,I_m(v)$, and $T_1(v,v'),\ldots,T_n(v,v')$ such that:

$$\begin{aligned}
&\mathit{init}(v) \rightarrow I_0(v),\\
&I_{k-1}(v) \wedge \mathit{stem}_k(v,v') \rightarrow I_k(v'), && \text{for each } k \in 1..m\\
&I_m(v) \wedge \mathit{loop}_1(v,v') \rightarrow T_1(v,v'),\\
&T_{k-1}(v,v') \wedge \mathit{loop}_k(v',v'') \rightarrow T_k(v,v''), && \text{for each } k \in 2..n\\
&\mathit{wf}(T_n(v,v')).
\end{aligned}$$

Note that the last clause, which is a unit clause, requires that the relation $T_n(v,v')$ is well-founded, i.e., does not admit any infinite chains.

**Search tree interpolation** Interpolation has been used for optimizing the search for solutions for a constraint programming goal [17]. In that work, it is considered the case when the search tree corresponds to the state space exploration of an imperative program in order to prove some safety property. A node from the tree is labeled with a formula $s(v)$ that is a symbolic representation for reachable states at a program point. The tree structure corresponds to program transitions, a node $n$ has as many children as the transitions starting at the program point corresponding to $n$ , i.e., $next_1(v,v'),\ldots,next_m(v,v')$ . To optimize the search, symbolic states are generalized by computing interpolants in post-order tree traversal. During the tree traversal, for a node $n$ , initially labeled $s_0$ , and having children with labels $s_1$ to $s_m$ , a generalized label of the node $n$ is computed as $I_1(v)\wedge\cdots\wedge I_m(v)$ and is subject to the following implications:

$$\begin{aligned} &s_0(v)\rightarrow I_1(v)\wedge\cdots\wedge I_m(v)\\ &I_k(v)\rightarrow(next_k(v,v')\rightarrow s_k(v')) \qquad \text{for each } k\in 1..m \end{aligned}$$

These implications correspond to the following recursion-free Horn clauses,

$$\begin{aligned} &s_0(v)\rightarrow I_k(v), \qquad &\text{for each } k\in 1..m\\ &I_k(v)\rightarrow(\exists v' : next_k(v,v')\rightarrow s_k(v')), \qquad &\text{for each } k\in 1..m \end{aligned}$$

where the quantifier elimination in $\exists v' : next_k(v,v')\rightarrow s_k(v')$ can be automated for $next_k$ and $s_k$ background constraints in the theory of linear arithmetic.

**Nested interpolation** For programs with procedures, interpolation can compute over-approximations of sets of program states that are expressed over variables that are in scope at respective program locations, see e.g. [14, 15]. A procedural program consists of a set of procedures $P$ including the main procedure *main*, global program variables $g$ that include a dedicated variable for return value passing, as well as procedure descriptions. For each procedure $p\in P$ we provide its local variables $l_p$, a finite set of intra-procedural program transitions of the form $inst^p(g,l_p,g',l'_p)$, a finite set of call transitions of the form $call^{p,q}(g,l_p,l_q)$ where $q\in P$ is the name of the callee, a finite set of return transitions of the form $ret^p(g,l_p,g')$, as well as a description of safe states $safe^p(g,l_p)$.

A path in a procedural program is a sequence of program transitions (including intra-procedural, call and return transitions) that respects the calling discipline, which we do not formalize here.

Given a path $next_1(v,v'),\ldots,next_n(v,v')$. Find $I_0(v_0),I_1(v_1),\ldots,I_n(v_n)$, where $v_0,\ldots,v_n$ are determined through the following implications, such that:

$$\begin{aligned} &init(g,l_{main})\rightarrow I_0(g,l_{main}),\\ &I_{k-1}(g,l_p)\wedge\begin{cases} inst^p(g,l_p,g',l'_p)\rightarrow I_k(g',l'_p), & \text{if } next_k(v,v')=inst^p(g,l_p,g',l'_p)\\ call^{p,q}(g,l_p,l_q)\rightarrow I_k(g,l_q), & \text{if } next_k(v,v')=call^{p,q}(g,l_p,l_q)\\ ret^p(g,l_p,g')\rightarrow I_k(g',l_q), & \text{if } next_k(v,v')=ret^p(g,l_p,g') \text{ returns to } q \end{cases}\\ &\text{for each } k\in 1..n\\ &I_n(g,l_p)\rightarrow safe^p(g,l_p), \quad \text{when } next_n(v,v') \text{ occurs in procedure } p. \end{aligned}$$

Similarly to the previously described interpolation problems, there are no recursive dependencies in the above clauses.

**State/transition interpolation** As illustrated by the example of well-founded interpolation, interpolants can represent over-approximations of sets of states as well as binary relations. The Whale algorithm provides a further example of such usage [1]. Given a sequence of assertions $next_1(v,v'),\ldots,next_n(v,v')$ that represent an under-approximation of a path through a procedure with a guard $g(v)$ and a summary $s(v,v')$. Find guards $G_1(v),\ldots,G_n(v)$ and summaries $S_1(v,v'),\ldots,S_n(v,v')$ such that:

$$\begin{array}{ll} next_k(v,v') \to S_k(v,v'), & \text{for each } k \in 1..n \\ g(v) \to G_1(v), & \\ G_k(v) \wedge S_k(v,v') \to G_{k+1}(v'), & \text{for each } k \in 1..n-1 \\ G_n(v) \wedge S_n(v,v') \to s(v,v'). & \end{array}$$

There are no recursive dependencies among the unknown guards and summaries.

**Solving unfoldings of recursive Horn clauses** A variety of reachability and termination verification problems for programs with procedures, multi-threaded programs, and functional programs can be formulated as the satisfiability of a set of recursive Horn clauses, e.g., [8, 11, 13]. These clauses are obtained from the program during a so-called constraint generation step. The satisfiability checking performed during the constraint solving step amounts to the inference of inductive invariants, procedure summaries, function types and other required auxiliary assertions. Existing solvers, e.g., HSF [8] and $\mu Z$ [16], rely on solving recursion-free unfoldings when iteratively constructing a solution for recursive Horn clauses.

We illustrate the generation of recursion-free unfolding using an invariance proof rule for flat programs. This rule can be formalised by as follows. For a given program find an invariant $Inv(v)$ such that

$$\begin{array}{ll} init(v) \to Inv(v), & \\ Inv(v) \wedge next(v,v') \to Inv(v'), & \text{for each program transition } next(v,v') \\ Inv(v) \to safe(v). & \end{array}$$

An unfolding of these recursive clauses introduces auxiliary relations that refer to $Inv(v)$ at each intermediate step. For example we consider an unfolding that starts with the first clause above and then applies a clause from the second line for a transition $next_1(v,v')$ and then for a transition $next_2(v,v')$ before traversing the last clause. This unfolding is represented by the following recursion-free clauses:

$$\begin{array}{l} init(v) \to Inv_0(v),\ Inv_0(v) \wedge next_1(v,v') \to Inv_1(v'), \\ Inv_1(v) \wedge next_2(v,v') \to Inv_2(v'),\ Inv_2(v) \to safe(v). \end{array}$$

A solution for these clauses contributes to solving the recursive clauses.

## 3 Algorithm overview

In this section we briefly describe how InterHorn solves recursion-free Horn clauses. We refer to [11, Section 7] for a solving algorithm for clauses over linear rational arithmetic, to [12] for a treatment of a combined theory of linear rational arithmetic and uninterpreted functions, and to [22] for a support of well-foundedness conditions.

InterHorn critically relies on the following two observations. First, applying resolution on clauses that describe the interpolation problem terminates and yields an assertion that does not contain any unknown relations. For example, resolution of clauses in Section 2 that describe path, transition, nested

and state/transition interpolation results in the implication of the form $init(v_0) \wedge (\bigwedge_{k=1}^{n} next_k(v_{k-1}, v_k)) \rightarrow safe(v_n)$. Second, the obtained assertion is valid if and only if the set of clauses is satisfiable. From the proof of validity (or alternatively, from the proof of unsatisfiability of the negated assertion) we construct the solutions.

**Clauses without well-foundedness conditions** INTERHORN goes through three main steps when given a set of recursion-free clauses that does not contain any well-foundedness condition. For example, we consider the following recursion-free clauses as input:

$$x \geq 10 \rightarrow p(x), \;\; p(u) \wedge w = u + v \rightarrow q(v,w), \;\; q(y,z) \wedge y \leq 0 \rightarrow z \geq y.$$

During the first step we apply resolution on the set of clauses. Since the clauses are recursion-free, the resolution application terminates. The result is an assertion that only contains constraints from the background theory. After applying resolution we obtain for our example (note that we use fresh variables here to stress the fact that clauses are implicitly universally quantified): $a \geq 10 \wedge c = a + b \wedge b \leq 0 \rightarrow c \geq b$.

The second step amounts to checking the validity of the obtained assertion.[1] If the assertion is not valid then we report that the original set of clauses imposes constraints that cannot be satisfied. Otherwise we produce a proof of validity. In our example the proof of validity can be represented as a weighted sum of the inequalities in the antecedent of the implication, with the weights 1, $-1$, and 0, respectively.

The third step traverses the input clauses and computes the solution assignment by taking the proof into account. For the clause $x \geq 10 \rightarrow p(x)$ we determine that $x \geq 10$ contributes to $p(x)$ with a weight 1, since during the resolution $x \geq 10$ gave rise to $a \geq 10$ whose weight is 1. Thus we obtain $p(x) = (x \geq 10)$. For the clause $p(u) \wedge w = u + v \rightarrow q(v,w)$ we combine $p(u)$ and $w = u + v$ with the weight of the latter set to $-1$, since $w = u + v$ yielded a contribution to the proof with weight $-1$. This leads to $q(v,w) = (u \geq 10) + (-1) * (w = u + v) = (w \geq 10 + v)$.

Finally, INTERHORN outputs the solution:

$$p(x) = (x \geq 10), \;\; q(v,w) = (w \geq 10 + v).$$

We observe that the substitution of the solutions into the input clauses produces valid implications: $x \geq 10 \rightarrow x \geq 10$, $u \geq 10 \wedge w = u + v \rightarrow w \geq 10 + v$, and $z \geq 10 + y \wedge y \leq 0 \rightarrow z \geq y$.

**Clauses with a well-foundedness condition** In case of a well-foundedness condition occurring in the input, INTERHORN introduces additional steps to take this condition into account. For example, we consider the following recursion-free clauses with a well-foundedness condition as input:

$$x \geq 10 \rightarrow p(x), \quad p(u) \wedge w = u + v \rightarrow q(v,w), \quad q(y,z) \wedge y \leq 0 \rightarrow r(y,z),$$
$$wf(r(s,t)).$$

The first step is again the resolution of the given clauses that produces a clause providing an under-approximation for the relation that is subject to the well-foundedness condition. For our example, we obtain: $a \geq 10 \wedge c = a + b \wedge b \leq 0 \rightarrow r(b,c)$.

The second step attempts to find a well-founded relation that over-approximates the projection of the antecedent of the clause obtained by resolution on the variables in its head. For our example this

[1] Instead of validity checking we can check satisfiability of the negated assertion.

projection amounts to performing an existential quantifier elimination on $\exists a : a \geq 10 \wedge c = a + b \wedge b \leq 0$, which gives $c \geq 10 + b \wedge b \leq 0$. This relation is well-founded, which is witnessed by a ranking relation over $b$ and $c$ with a bound component $b \leq 0$ and the decrease component $c \geq b + 1$.

The third step uses the well-founded over-approximation to construct a clause that introduces an upper bound on the relation under well-foundedness condition. This clause replaces the well-formedness condition by an approximation condition wrt. an assertion. For our example, the clause $wf(r(s,t))$ is replaced by the clause $r(s,t) \rightarrow (s \leq 0 \wedge t \geq s + 1)$.

Lastly, we apply the solving method for clauses without well-foundedness conditions described previously. In our example, the set of clauses to be solved becomes:

$$x \geq 10 \rightarrow p(x), \quad p(u) \wedge w = u + v \rightarrow q(v,w), \quad q(y,z) \wedge y \leq 0 \rightarrow r(y,z),$$
$$r(s,t) \rightarrow (s \leq 0 \wedge t \geq s + 1).$$

Finally, InterHorn outputs the solution:

$$p(x) = (x \geq 10), \;\; q(v,w) = (w \geq 10 + v), \;\; r(s,t) = (s \leq 0 \wedge t \geq s + 10).$$

## 4 Implementation

InterHorn is implemented in SICStus Prolog [25]. For computing proofs of validity (resp. unsatisfiability) over linear rational arithmetic theory, InterHorn relies on a proof producing version of a simplex algorithm [10]. For computing well-founded approximations (also over linear rational arithmetic theory), InterHorn uses a linear ranking functions synthesis algorithm [21]. InterHorn can be downloaded from `http://www7.in.tum.de/tools/interhorn/`, accepts input in form of Prolog terms and outputs an appropriately formatted result.

## 5 Conclusion

We presented InterHorn, a solver for recursion-free Horn clauses that can be used to deal with various interpolation problems. The main directions for the future development include adding support for uninterpreted functions, along the lines of [12], and integer arithmetic. After developing our work, we became aware of a related work highlighting the relation between interpolation and recursion-free Horn clauses [23]. The authors of [23] show that some interpolation problems correspond to various fragments of recursion-free Horn clauses and establish complexity results for these fragments assuming the background theory of linear integer arithmetic. Our work is less concerned with the different fragments of recursion-free Horn clauses and more with how interpolation problems arise in software verification. The well-founded interpolation problem is beyond the scope of [23].

**Acknowledgements**

This research was supported in part by ERC project 308125 VeriSynth, by Austrian Science Fund NFN RiSE (Rigorous Systems Engineering), and by the ERC Advanced Grant QUAREM (Quantitative Reactive Modeling).

## References


[1] Aws Albarghouthi, Arie Gurfinkel & Marsha Chechik (2012): *Whale: An Interpolation-Based Algorithm for Inter-procedural Verification*. In: *VMCAI*, doi:10.1007/978-3-642-27940-9_4.

[2] Aws Albarghouthi, Yi Li, Arie Gurfinkel & Marsha Chechik (2012): *Ufo: A Framework for Abstraction- and Interpolation-Based Software Verification*. In: *CAV*, doi:10.1007/978-3-642-31424-7_48.

[3] Angelo Brillout, Daniel Kroening, Philipp Rümmer & Thomas Wahl (2010): *An Interpolating Sequent Calculus for Quantifier-Free Presburger Arithmetic*. In: *IJCAR*, doi:10.1007/978-3-642-14203-1_33.

[4] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina & Aliaksei Tsitovich (2010): *The OpenSMT Solver*. In: *TACAS*, doi:10.1007/978-3-642-12002-2_12.

[5] Jürgen Christ, Jochen Hoenicke & Alexander Nutz (2012): *SMTInterpol: An Interpolating SMT Solver*. In: *SPIN*, doi:10.1007/978-3-642-31759-0_19.

[6] Byron Cook, Andreas Podelski & Andrey Rybalchenko (2005): *Abstraction Refinement for Termination*. In: *SAS*, doi:10.1007/11547662_8.

[7] Sergey Grebenshchikov, Ashutosh Gupta, Nuno P. Lopes, Corneliu Popeea & Andrey Rybalchenko (2012): *HSF(C): A Software Verifier Based on Horn Clauses - (Competition Contribution)*. In: *TACAS*, doi:10.1007/978-3-642-28756-5_46.

[8] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea & Andrey Rybalchenko (2012): *Synthesizing software verifiers from proof rules*. In: *PLDI*, doi:10.1145/2254064.2254112.

[9] Alberto Griggio, Thi Thieu Hoa Le & Roberto Sebastiani (2011): *Efficient Interpolant Generation in Satisfiability Modulo Linear Integer Arithmetic*. In: *TACAS*, doi:10.1007/978-3-642-19835-9_13.

[10] Ashutosh Gupta (2011): *Constraint solving for verification*. Ph.D. thesis, TUM.

[11] Ashutosh Gupta, Corneliu Popeea & Andrey Rybalchenko (2011): *Predicate abstraction and refinement for verifying multi-threaded programs*. In: *POPL*, doi:10.1145/1926385.1926424.

[12] Ashutosh Gupta, Corneliu Popeea & Andrey Rybalchenko (2011): *Solving Recursion-Free Horn Clauses over LI+UIF*. In: *APLAS*, doi:10.1007/978-3-642-25318-8_16.

[13] Ashutosh Gupta, Corneliu Popeea & Andrey Rybalchenko (2011): *Threader: A Constraint-Based Verifier for Multi-threaded Programs*. In: *CAV*, doi:10.1007/978-3-642-22110-1_32.

[14] Matthias Heizmann, Jochen Hoenicke & Andreas Podelski (2010): *Nested interpolants*. In: *POPL*, doi:10.1145/1706299.1706353.

[15] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar & Kenneth L. McMillan (2004): *Abstractions from proofs*. In: *POPL*, doi:10.1145/964001.964021.

[16] Krystof Hoder & Nikolaj Bjørner (2012): *Generalized Property Directed Reachability*. In: *SAT*, doi:10.1007/978-3-642-31612-8_13.

[17] Joxan Jaffar, Andrew E. Santosa & Razvan Voicu (2009): *An Interpolation Method for CLP Traversal*. In: *CP*, pp. 454–469, doi:10.1007/978-3-642-04244-7_37.

[18] Ranjit Jhala & Kenneth L. McMillan (2005): *Interpolant-Based Transition Relation Approximation*. In: *CAV*, doi:10.1007/11513988_6.

[19] Daniel Kroening & Georg Weissenbacher (2011): *Interpolation-Based Software Verification with Wolverine*. In: *CAV*, doi:10.1007/978-3-642-22110-1_45.

[20] Kenneth L. McMillan (2006): *Lazy Abstraction with Interpolants*. In: *CAV*, doi:10.1007/11817963_14.

[21] Andreas Podelski & Andrey Rybalchenko (2004): *A Complete Method for the Synthesis of Linear Ranking Functions*. In: *VMCAI*, doi:10.1007/978-3-540-24622-0_20.

[22] Corneliu Popeea & Andrey Rybalchenko (2012): *Compositional Termination Proofs for Multi-threaded Programs*. In: *TACAS*, doi:10.1007/978-3-642-28756-5_17.

[23] Philipp Rümmer, Hossein Hojjat & Viktor Kuncak (2013): *Classifying and Solving Horn clauses for verification*. In: *VSTTE*, doi:10.1007/978-3-642-54108-7_1.

[24] Ondrej Sery, Grigory Fedyukovich & Natasha Sharygina (2012): *FunFrog: Bounded Model Checking with Interpolation-Based Function Summarization*. In: *ATVA*, doi:10.1007/978-3-642-33386-6_17.

[25] The Intelligent Systems Laboratory (2001): *SICStus Prolog User's Manual*. Swedish Institute of Computer Science. Release 3.8.7.