Inferring Simple Solutions to Recursion-free Horn Clauses via Sampling*

Hiroshi Unno[1] and Tachio Terauchi[2]

[1] University of Tsukuba
uhiro@cs.tsukuba.ac.jp
[2] JAIST
terauchi@jaist.ac.jp

Abstract. Recursion-free Horn-clause constraints have received much recent attention in the verification community. It extends Craig interpolation, and is proposed as a unifying formalism for expressing abstraction refinement. In abstraction refinement, it is often desirable to infer "simple" refinements, and researchers have studied techniques for inferring simple Craig interpolants. Drawing on the line of work, this paper presents a technique for inferring simple solutions to recursion-free Horn-clause constraints. Our contribution is a constraint solving algorithm that lazily samples fragments of the given constraints whose solution spaces are used to form a simple solution for the whole. We have implemented a prototype of the constraint solving algorithm in a verification tool, and have confirmed that it is able to infer simple solutions that aid the verification process.

1 Introduction

In program verification, Craig interpolation [3] is a technique for discovering predicates that can be used to prove the correctness of the given program. For example, in predicate abstraction, interpolants from the formula representing the counterexample are used as predicates to refute the counterexample [7], and in lazy abstraction via interpolation, interpolants from the formula representing the program unwinding are used to construct sufficient loop invariants [12]. In general, there is more than one interpolant that can be inferred from the same formula, and which interpolant is inferred can significantly affect the performance of the client verifier. The "goodness" of an interpolant is an elusive characteristic, and while there is not yet a definite measure, it has been suggested that *simple* interpolants often work better (perhaps justified by the belief that correct programs tend to be correct for simple reasons, per Occam's razor). Recently, researchers have proposed to infer simple interpolants between a pair of formulas by *sampling* conjunctions of atoms from each formula, inferring their interpolant, and repeating the process until the interpolant for the whole

* This work was supported by MEXT Kakenhi 23220001, 26330082, 25280023, and 25730035.

is found [17, 1]. By inferring simple interpolants for the samples that are likely to generalize, the method efficiently infers a simple interpolant for the whole.

In this paper, we extend the idea to inferring simple solution to recursion-free Horn-clause constraints. Recently, recursion-free Horn-clause constraints have received much attention in the verification community as they generalize interpolation and can express the predicate discovery process of a wide variety of software verifiers (imperative, procedural, higher-order functional, concurrent, etc. [20, 18, 6, 5, 4, 14, 2, 19]).[3] We emphasize that inferring a simple solution to recursion-free Horn-clause constraints is non-trivial and *cannot be done by simply applying the methods for interpolation*, because one must look simultaneously for simple predicates to be assigned to each predicate variable in the given constraints that together satisfy the constraints (e.g., it cannot be done by just iteratively applying interpolation as a blackbox process [20, 18]).

The key ideas in our approach are to 1.) maintain as samples *conjunctive* recursion-free Horn-clause constraint that only contain clauses whose formula part is a conjunction of atoms, 2.) infer a simple solution to the samples via a novel decompositional approach (cf. Sections 3.1–3.3), and 3.) check if the solution inferred for the samples is also a solution for the whole, and if not, obtain a new sample as a counterexample and repeat the process. Finally, 4.) instead of computing a concrete solution for each subproblem, we compute an *abstract solution space* representing a possibly infinite set of solutions, thereby making the process more likely to be able to find a simple solution for the whole.

Related Work. Besides the above sampling-based approaches to inferring simple interpolants that inspired this work, previous research has proposed to infer simple interpolants by post-processing the proof (of $\models \phi_1 \Rightarrow \phi_2$) in a proof-based interpolation [8].

To our knowledge, this paper is the first work on inferring simple solutions to recursion-free Horn-clause constraints. Existing approaches to solving recursion-free Horn-clause constraints can be classified into two types: the *iterative approach* that uses interpolation as a blackbox process to solve the constraints one predicate variable at a time [20, 18], and the *constraint-expansion approach* that reduces the problem to tree interpolation (equivalently, solving "tree-like" constraints) [11, 13, 14]. As remarked above, the iterative approach is unsuited for inferring simple solutions because a solution inferred for one predicate variable can affect the rest and block the discovery of a simple solution for the whole. The constraint-expansion approach is also unsuited for inferring simple solutions because it makes exponentially many copies of predicate variables whose solutions are conjuncted to form the solution for the original, thereby resulting in a complex solution (see also the discussion in Section 3.1).

Paper Organization. The rest of the paper is organized as follows. Section 2 presents preliminary definitions. Section 3 and its subsections describe the new constraint solving algorithm in a top-down manner. We first present the top-

[3] Interpolation between ϕ_1 and ϕ_2 is equivalent to solving the Horn-clause constraint $\{P(\widetilde{x}) \Leftarrow \phi_1, \bot \Leftarrow P(\widetilde{x}) \wedge \neg\phi_2\}$ where $\{\widetilde{x}\} = \mathit{fvs}(\phi_1) \cap \mathit{fvs}(\phi_2)$.

level process in Section 3. Section 3.1 describes the sub-algorithm for inferring simple solutions for samples. As we explain there in more detail, inferring simple solutions to samples requires its own innovations as simply applying the existing approaches can produce complex solutions. To this end, we present a novel approach where the problem is decomposed into smaller subproblems for which simple solutions can be found easily and combined to form a simple solution for the whole sample set. We describe the approach in detail in Sections 3.1–3.3. We report on a preliminary implementation and experiment results in Section 4, and conclude the paper in Section 5. The extended report [21] contains extra materials and omitted proofs.

2 Preliminaries

A *formula* ϕ in the signature of quantifier-free linear rational arithmetic (QFLRA) is a Boolean combination of atoms. An *atom* (or *literal*) p is an inequality of the form $t_1 \geq t_2$ or $t_1 > t_2$ where t_i are terms. A *term* t is either a *variable* x, a *rational constant* r, a multiplication of a term by a rational constant $r \cdot t$, or a summation of terms $t_1 + t_2$. We write $\bot$ and $\top$ respectively for contradiction and tautology. A *predicate variable application* a is of the form $P(\widetilde{t})$ where P is a *predicate variable* of the arity $|\widetilde{t}|$. We write $ar(P)$ for the arity of P.

A *Horn clause* (or simply *clause*) hc is defined to be of the form $a_0 \Leftarrow a_1 \wedge \cdots \wedge a_n \wedge \phi$. We call a_0 (resp. $a_1 \wedge \cdots \wedge a_n \wedge \phi$) the *head* (resp. *body*) of hc. We write $fvs(hc)$ (resp. $fvs(\phi)$) for the set of term variables in hc (resp. ϕ). We write $pvL(hc)$ for the predicate variable occurring on the left hand side of $\Leftarrow$ and $pvsR(hc)$ for the set of predicate variables occurring in the right hand side of $\Leftarrow$. We write $pvs(hc)$ for the set of predicate variables occurring in hc (i.e., $pvs(hc) = \{pvL(hc)\} \cup pvsR(hc)$).

We define a *Horn clause constraint set* (HCCS) to be a pair $(\mathcal{H}, P_\bot)$ where $\mathcal{H}$ is a finite set of clauses and $P_\bot$ is a predicate variable in $\mathcal{H}$ with $ar(P_\bot) = 0$ (intuitively, $P_\bot$ is implicitly constrained by the clause $\bot \Leftarrow P_\bot()$). We define $fvs(\mathcal{H}) = \bigcup_{hc \in \mathcal{H}} fvs(hc)$. We define $pvs(\mathcal{H}) = \bigcup_{hc \in \mathcal{H}} pvs(hc)$, $pvsL(\mathcal{H}) = \{pvL(hc) \mid hc \in \mathcal{H}\}$, $pvsR(\mathcal{H}) = \bigcup_{hc \in \mathcal{H}} pvsR(hc)$, $roots(\mathcal{H}) = pvsL(\mathcal{H}) \backslash pvsR(\mathcal{H})$, $inters(\mathcal{H}) = pvsL(\mathcal{H}) \cap pvsR(\mathcal{H})$, and $leaves(\mathcal{H}) = pvsR(\mathcal{H}) \setminus pvsL(\mathcal{H})$. We say that $\mathcal{H}$ is *single-root* if $roots(\mathcal{H})$ is singleton, and write $root(\mathcal{H})$ for P such that $\{P\} = roots(\mathcal{H})$.

Concrete and Abstract Solutions. A predicate substitution θ is a finite map from predicate variables P to predicates of the form $\lambda(x_1, \ldots, x_{ar(P)}).\phi$ such that $fvs(\phi) \subseteq \{x_1, \ldots, x_{ar(P)}\}$. Given an HCCS $(\mathcal{H}, P_\bot)$, a predicate substitution θ with $pvs(\mathcal{H}) \subseteq dom(\theta)$ is called a *solution* of $\mathcal{H}$ if $\models \theta(hc)$ for each $hc \in \mathcal{H}$ and $\models \bot \Leftarrow \theta(P_\bot)()$. We write $\theta \models (\mathcal{H}, P_\bot)$ when θ is a solution of $(\mathcal{H}, P_\bot)$.

We define *abstract solution space* that represents a possibly infinite set of solutions. To this end, we define *formula template* ψ to be a formula but with the grammar extended to include terms of the form $c \cdot t$ where c is an *unknown coefficient variable*. We define an *abstract solution space* S to be a pair (Θ, ϕ) where Θ is a finite map from predicate variables P to *predicate templates* of

the form $\lambda(x_1, \ldots, x_{ar(P)}).\psi$ and ϕ is a (non-template) QFLRA formula over unknowns. For $S = (\Theta, \phi)$ and $P \in dom(\Theta)$, we write $S(P)$ for $\Theta(P)$. We say that a concrete solution θ is an *instance* of an abstract solution space (Θ, ϕ), written $\theta \succeq (\Theta, \phi)$, if $dom(\theta) = dom(\Theta)$ and there exists a map σ from unknowns to rationals such that $\models \sigma(\phi)$ and for all $P \in dom(\theta)$, $\theta(P) = \sigma(\Theta(P))$. We write $S' \succeq S$ if for all θ, $\theta \succeq S'$ implies $\theta \succeq S$. We write $S' \succeq_P S$ if $S' \succeq S$ and $S'(P)$ contains no unknowns.

Horn Clause Constraint Kinds. The *dependency relation* $\lhd_{\mathcal{H}}$ is defined to be the relation that, for all $P, Q \in pvs(\mathcal{H})$, $Q \lhd_{\mathcal{H}} P$ if and only if $Q(\widetilde{t_1}) \Leftarrow a \wedge \cdots \wedge P(\widetilde{t_2}) \wedge \cdots \wedge \phi \in \mathcal{H}$. We write $\lhd^*_{\mathcal{H}}$ for the reflexive transitive closure of $\lhd_{\mathcal{H}}$ and $\lhd^+_{\mathcal{H}}$ for the transitive closure of $\lhd_{\mathcal{H}}$. We say that P is *recursive* if $P \lhd^+_{\mathcal{H}} P$. We say that P is *head-joining* (resp. *body-joining*) if P occurs more than once in the left (resp. right) hand sides of clauses in $\mathcal{H}$. We write *recpvs*($\mathcal{H}$), *hjnpvs*($\mathcal{H}$), and *bjnpvs*($\mathcal{H}$) respectively for the set of recursive, head-joining, and body-joining predicate variables in $\mathcal{H}$. We say that $\mathcal{H}$ is *recursion-free* if *recpvs*($\mathcal{H}$) $= \emptyset$, is *body-disjoint* if *bjnpvs*($\mathcal{H}$) $= \emptyset$, and is *head-disjoint* if *hjnpvs*($\mathcal{H}$) $= \emptyset$.[4] We say that $\mathcal{H}$ is *conjunctive* if for each $hc \in \mathcal{H}$, the formula part of hc is a conjunction of literals. We say that a single-root $\mathcal{H}$ is *connected* if for any $P \in pvs(\mathcal{H})$, $root(\mathcal{H}) \lhd^*_{\mathcal{H}} P$. We extend the notions to HCCSs in the obvious way (e.g., $(\mathcal{H}, P_\bot)$ is recursion-free if $\mathcal{H}$ is recursion-free).

Any Horn clause set in this paper will be recursion-free. Therefore, in what follows, we restrict ourselves to recursion-free Horn clause sets and HCCSs and omit the redundant qualifier "recursion-free".

Example 1. Consider the HCCS $(\mathcal{H}_{ex1}, P_\bot)$ where $\mathcal{H}_{ex1}$ is the set of clauses below.

$$\begin{aligned} P(x, y, z) &\Leftarrow x \geq z \wedge y \geq 2 - z \\ Q(x, y) &\Leftarrow P(x, y, z) \wedge (z = 0 \vee z = 1 \vee z = 2) \\ P_\bot() &\Leftarrow Q(x, y) \wedge Q(-x, -y) \end{aligned}$$

We have *recpvs*($\mathcal{H}_{ex1}$) $= \emptyset$, *hjnpvs*($\mathcal{H}_{ex1}$) $= \emptyset$, and *bjnpvs*($\mathcal{H}_{ex1}$) $= \{Q\}$, and so the HCCS is recursion-free and head-disjoint but neither conjunctive nor body-disjoint. (An equality $t_1 = t_2$ is $t_1 \geq t_2 \wedge t_2 \geq t_1$.)

A solution for the HCCS, θ_{ex1}, and an abstract solution space for the HCCS, $(\Theta_{ex1}, \phi_{ex1})$ are shown below. Note that $\theta_{ex1} \succeq (\Theta_{ex1}, \phi_{ex1})$.

$$\begin{aligned} \theta_{ex1}(P) &= \lambda(x, y, z).x + y \geq 2 \\ \theta_{ex1}(Q) &= \lambda(x, y).x + y \geq 1 \\ \theta_{ex1}(P_\bot) &= \lambda().\bot \end{aligned} \qquad \begin{aligned} \Theta_{ex1}(P) &= \lambda(x, y, z).c_0 + c_1 \cdot x + c_2 \cdot y \geq 0 \\ \Theta_{ex1}(Q) &= \lambda(x, y).x + y \geq 1 \\ \Theta_{ex1}(P_\bot) &= \lambda().\bot \\ \phi_{ex1} &\equiv 0 < c_1 = c_2 \leq -c_o \leq 2 \cdot c_1 \end{aligned}$$

3 The Top-Level Procedure

Figure 1 shows the top-level procedure of the constraint solving algorithm $\mathcal{A}_{solve}$ which takes as input an HCCS $(\mathcal{H}, P_\bot)$ and returns its solution or detects that it

[4] The terminologies are adopted from [14, 13].

is unsolvable. As remarked in Section 1, the algorithm looks for a simple solution of the given HCCS by lazy sampling. $\mathcal{A}_{solve}$ initializes the sample set *Samples* to $\emptyset$ (line 2), and repeats the loop (lines 3-12) until convergence. The loop first calls the sub-algorithm $\mathcal{A}_{samp}$ on the HCCS $(\mathit{Samples}, P_\bot)$ to find an abstract space of solutions to the current sample set. If no solution is found for the samples, then no solution exists for the whole constraint set $(\mathcal{H}, P_\bot)$ either, and we exit the loop (line 5). Otherwise, an abstract solution space S for the samples is inferred, and we pick a concrete instance θ of S (line 7) as the *candidate* solution. If θ is a solution for the whole then we return it as the inferred solution (line 8). Otherwise, there is a clause in $\mathcal{H}$, say $P(\widetilde{t}) \Leftarrow \bigwedge \widetilde{a} \wedge \phi$, that is unsatisfied and a model σ in which the clause is invalid with θ. From the clause and σ, we obtain the conjunctive clause $P(\widetilde{t}) \Leftarrow \bigwedge \widetilde{a} \wedge \bigwedge C(\phi, \sigma)$ as the new sample to be added to the sample set (line 12). Here, $C(\phi, \sigma)$ is the set of atoms representing the part of ϕ where σ holds true, and is defined as follows.

$$C(\phi, \sigma) = \{p \mid p \text{ occurs in } \phi \text{ and } \sigma \models p\} \cup \{\neg p \mid p \text{ occurs in } \phi \text{ and } \sigma \models \neg p\}$$

Intuitively, the added sample clause represents a portion of the input HCCS that is not yet covered by the solution found for the current sample set.

By construction, the sample HCCS $(\mathit{Samples}, P_\bot)$ is always conjunctive. The sub-algorithm $\mathcal{A}_{samp}$, whose details are deferred to Section 3.1, takes the conjunctive HCCS $(\mathit{Samples}, P_\bot)$ and infers an abstract space of solutions for it. Next, we show the correctness of $\mathcal{A}_{solve}$, assuming that $\mathcal{A}_{samp}$ works correctly (i.e., it returns a non-empty abstract solution space to the input conjunctive HCCS if it is solvable and otherwise returns *NoSol*). Let $D(\phi) = \{C(\phi, \sigma) \mid \sigma \models \phi\}$. Let $(D(\mathcal{H}), P_\bot)$ be the conjunctive HCCS obtained by replacing each clause $a \Leftarrow \bigwedge \widetilde{a} \wedge \phi$ in $\mathcal{H}$ with the clauses $\{a \Leftarrow \bigwedge \widetilde{a} \wedge \bigwedge C \mid C \in D(\phi)\}$. Note that, because $D(\phi)$ is finite, $D(\mathcal{H})$ is also finite and $(D(\mathcal{H}), P_\bot)$ is an HCCS. Also, the following can be shown from the fact that $\models \phi \Leftrightarrow \bigvee_{C \in D(\phi)} \bigwedge C$.

```
01: A_solve((H, P_⊥)) =
02:   Samples := ∅;
03:   while true do
04:     match A_samp((Samples, P_⊥)) with
05:         NoSol → return NoSol
06:       | Sol(S) →
07:         let θ ≽ S in
08:         if θ ⊨ (H, P_⊥) then
09:           return Sol(θ)
10:         else
11:           let σ, P(t̃) ⇐ ⋀ã ∧ φ ∈ H
                where σ ⊭ θ(⋀ã ∧ φ ⇒ P(t̃)) in
12:           Samples :=
                Samples ∪ {P(t̃) ⇐ ⋀ã ∧ ⋀C(φ, σ)}
```

Fig. 1. The Top-Level Procedure

Lemma 1. *θ is a solution of $(\mathcal{H}, P_\bot)$ if and only if it is a solution of $(D(\mathcal{H}), P_\bot)$.*

We can also show that, in each loop iteration, the added sample is not in the current sample set, and therefore the sample set grows monotonically as the loop progresses, as stated in the following lemma.

Lemma 2. *Suppose $\theta \models (\mathit{Samples}, P_\bot)$, $\sigma \models \bigwedge \theta(\widetilde{a}) \wedge \phi$, and $\sigma \not\models \theta(P)(\widetilde{t})$. Then, $P(\widetilde{t}) \Leftarrow \widetilde{a} \wedge \bigwedge C(\phi, \sigma) \notin \mathit{Samples}$.*

From the lemmas, we show the correctness of $\mathcal{A}_{solve}$, stated in the theorem below.

Theorem 1 (Correctness of $\mathcal{A}_{solve}$). *Given an HCCS $(\mathcal{H}, P_\perp)$, $\mathcal{A}_{solve}((\mathcal{H}, P_\perp))$ returns a solution of $(\mathcal{H}, P_\perp)$ if $(\mathcal{H}, P_\perp)$ is solvable, and otherwise returns NoSol.*

A reader may wonder why $\mathcal{A}_{solve}$ does not directly check if there exists a solution to the input HCCS from the entire abstract solution space S returned by $\mathcal{A}_{samp}$ (i.e., check $\exists\theta \succeq S.\theta \models (\mathcal{H}, P_\perp)$) and infer new samples by using the entire S if not. We opt against the approach because checking $\exists\theta \succeq S.\theta \models (\mathcal{H}, P_\perp)$ requires an expensive non-linear constraint solving. Instead, we let $\mathcal{A}_{solve}$ choose a concrete solution from S to be used as a candidate.[5]

Example 2. Consider running $\mathcal{A}_{solve}$ on the HCCS $(\mathcal{H}_{ex1}, P_\perp)$ from Example 1. Suppose that at some iteration, $\mathit{Samples} = \{P(x,y,z) \Leftarrow x \geq z \wedge y \geq 2 - z, Q(x,y) \Leftarrow P(x,y,z) \wedge z = 0, P_\perp() \Leftarrow Q(x,y) \wedge Q(-x,-y)\}$, and $\mathcal{A}_{samp}$ returned some abstract solution space S given $(\mathit{Samples}, P_\perp)$.

Let $\theta \succeq S$ be the candidate solution chosen at line 7 where $\theta = \{P \mapsto \lambda(x,y,z).y \geq 2 - z, Q \mapsto \lambda(x,y).y \geq 2, P_\perp \mapsto \lambda().\bot\}$. Because $\theta \not\models (\mathcal{H}_{ex1}, P_\perp)$, we obtain a new sample. A possible sample obtained here is $Q(x,y) \Leftarrow P(x,y,z) \wedge z = 2$. Adding the new sample to *Samples*, in the next loop iteration, as we shall detail in Example 3, $\mathcal{A}_{samp}$ returns an abstract solution space containing the solution θ_{ex1} shown in Example 1. ▲

3.1 The Sub-Algorithm $\mathcal{A}_{samp}$

$\mathcal{A}_{samp}$ takes as input a conjunctive HCCS, and returns a non-empty abstract space of its solutions if it is solvable and otherwise returns *NoSol*. As remarked before, $\mathcal{A}_{samp}$ looks for *simple* solutions that are likely to generalize when given to the upper-procedure $\mathcal{A}_{solve}$ to be used as a candidate solution for the whole.

```
01: A_samp((H, P⊥)) =
02:   S := ({P⊥ ↦ λ().⊥} ∪ {P ↦ λx̃.⊤ | P ∈ Q}, ⊤)
        where Q = roots(H) \ {P⊥};
03:   WorkSet := initWS(H);
04:   while WorkSet ≠ ∅ do
05:     let H' ∈ WorkSet
          where root(H') ∉ ⋃_{H ∈ WorkSet} pvsR(H) in
06:     WorkSet := WorkSet \ {H'};
07:     let S' ≥_{root(H')} S in S := S';
08:     let C, LMap = MkCnsts(H', S, H) in
09:     for each (H'', P'⊥) ∈ C do
10:       match A_hj((H'', P'⊥)) with
11:           NoSol → return NoSol
12:         | Sol(S') →
13:             S := combSol_∧(S', S, LMap)
14:   return Sol(S)
```

Fig. 2. The Sub-Algorithm $\mathcal{A}_{samp}$

The internal workings of $\mathcal{A}_{samp}$ are quite intricate. The subtlety comes from body-joining predicate variables and head-joining predicate variables. Indeed, as we shall show in Section 3.3, inferring

[5] Perhaps a somewhat subtle aspect of $\mathcal{A}_{solve}$ is that it is guaranteed to terminate and return a correct result despite only considering one concrete solution from the set of solutions returned by $\mathcal{A}_{samp}$ in each iteration.

a simple solution to a conjunctive body-and-head-disjoint HCCS is easy in that such an HCCS has either no solution or a simple solution where each predicate contains just one atom. $\mathcal{A}_{samp}$ decomposes the problem into easily solvable parts and combines their solutions to obtain a simple solution for the whole. The key to the success is to do a *coarse* decomposition so that there are few subproblems to be solved and the solutions to be combined, thereby resulting in a simple solution for the whole sample set.

Figure 2 shows the overview of $\mathcal{A}_{samp}$. Given the input conjunctive HCCS $(\mathcal{H}, P_\bot)$, we initialize the abstract solution space S to map $P_\bot$ to $\lambda().\bot$ and the other root predicate variables $P \in \mathit{roots}(\mathcal{H}) \setminus \{P_\bot\}$ to $\lambda(x_1, \ldots, x_{ar(P)}).\top$ (line 2), and initialize the work set *WorkSet* to $\mathit{initWS}(\mathcal{H})$ which is the coarsest connected sets of clauses that partition $\mathcal{H}$ and are body-joined only at the roots and the leaves (informally, $\mathit{initWS}(\mathcal{H})$ partitions $\mathcal{H}$ into body-disjoint "trees"). Formally, $\mathit{initWS}(\mathcal{H}) = \{\{hc \in \mathcal{H} \mid P \blacktriangleleft^*_{(\mathcal{H}, \mathcal{R}\setminus\{P\})} \mathit{pvL}(hc)\} \mid P \in \mathcal{R}\}$ where $\mathcal{R} = (\mathit{bjnpvs}(\mathcal{H}) \cap \mathit{pvsL}(\mathcal{H})) \cup \mathit{roots}(\mathcal{H})$, and $Q \blacktriangleleft_{(\mathcal{H},\mathcal{R}')} R$ if and only if $Q \triangleleft_{\mathcal{H}} R$ and $Q \notin \mathcal{R}'$. As we show in the lemma below, $\mathit{initWS}(\mathcal{H})$ is indeed the coarsest connected partition of $\mathcal{H}$ that is body-joined only at the roots and the leaves.

Lemma 3. *$\mathit{initWS}(\mathcal{H})$ is the smallest set X that satisfies: 1. $\mathcal{H} = \bigcup X$, 2. $\forall \mathcal{H}_1, \mathcal{H}_2 \in X. \mathcal{H}_1 \cap \mathcal{H}_2 = \emptyset$, 3. $\forall \mathcal{H}' \in X.$ $\mathcal{H}'$ is connected, and 4. $\forall \mathcal{H}' \in X. \mathit{bjnpvs}(\mathcal{H}) \cap \mathit{inters}(\mathcal{H}') = \emptyset$.*

Then, we solve each element of *WorkSet* by calling $\mathcal{A}_{hj}$, starting from the root-most one that contains $P_\bot$, and recording the inferred solutions in S (lines 4-13). $\mathcal{A}_{hj}$ is a sub-algorithm that, given a conjunctive body-disjoint (but possibly head-joined) HCCS, infers its solution if it is solvable and otherwise returns *NoSol*. The detailed description of $\mathcal{A}_{hj}$ is deferred to Section 3.2.

To invoke $\mathcal{A}_{hj}$ on an element $\mathcal{H}' \in \mathit{WorkSet}$, $\mathcal{A}_{samp}$ first partially concretizes the current abstract solution space S so that it maps $\mathit{root}(\mathcal{H}')$ to a concrete predicate (line 7), and then uses *MkCnsts* to convert $\mathcal{H}'$ into the set of the conjunctive body-disjoint HCCSs $\mathcal{C}$ (line 8). *MkCnsts* also returns *LMap* that maps the copied leaf predicate variables in $\mathcal{C}$ to the originals in $\mathcal{H}'$. Formally, $\mathit{MkCnsts}(\mathcal{H}', S, \mathcal{H})$ constructs $\mathcal{C}$ and *LMap* as follows. Let $\mathcal{H}'_{lcpy}$ be $\mathcal{H}'$ with each leaf predicate variable application $P(\widetilde{t})$ replaced by $P_{cpy}(\widetilde{t})$ for a fresh predicate variable P_{cpy}. *LMap* is the map from the fresh predicate variable P_{cpy} to the original P that it replaced. Let $P_{rt} = \mathit{root}(\mathcal{H}')$ and $S(P_{rt}) = \lambda\widetilde{x}.\neg \bigvee_{i=1}^{n} \phi_i$ where each ϕ_i is a conjunction of literals. Then, $\mathcal{C}$ is the set of HCCSs $\{(\mathcal{H}'_{lcpy} \cup \mathcal{H}_{lcsts} \cup \{P'_\bot() \Leftarrow P_{rt}(\widetilde{x}) \wedge \phi_i\}, P'_\bot) \mid i \in \{1, \ldots, n\}\}$ where $P'_\bot$ is a fresh predicate variable and $\mathcal{H}_{lcsts}$ is the set of clauses below.

$$\{P(\widetilde{x}) \Leftarrow \phi_i \mid P \in \mathit{dom}(\mathit{LMap}) \\ \mathit{lsol}(\mathcal{H}, \mathit{LMap}(P)) = \lambda\widetilde{x}. \textstyle\bigvee_{i=1}^{m} \phi_i \text{ where each } \phi_i \text{ is conjunction of literals}\}$$

Here, $\mathit{lsol}(\mathcal{H}, P)$ is the predicate expressing the "lower-bound" solution of P that is implied by $\mathcal{H}$, and it is defined recursively as follows.

$$\mathit{lsol}(\mathcal{H}, P) = \lambda\widetilde{x}. \bigvee \left\{\phi \wedge \textstyle\bigwedge_{i=1}^{m} \mathit{lsol}(\mathcal{H}, R_i)(\widetilde{t_i}) \mid P(\widetilde{x}) \Leftarrow \phi \wedge \bigwedge_{i=1}^{m} R_i(\widetilde{t_i}) \in \mathcal{H}\right\}$$

Intuitively, $MkCnsts(\mathcal{H}', S, \mathcal{H})$ substitutes the solution $S(root(\mathcal{H}'))$ for $root(\mathcal{H}')$ in $\mathcal{H}'$, adds the constraints required for the leaf predicate variables, and expands the constraint so that the result is a set of conjunctive body-disjoint HCCSs.

The solution inferred for each constraint in $\mathcal{C}$ is combined and recorded in the abstract solution space S (line 12). The solution combination operation $combSol_\wedge$ combines the abstract solutions by conjuncting the constraints over the unknowns and conjuncting the predicate templates point-wise, using $LMap$ to conjunct the solutions for the copied leaf predicates into the original. Formally, $combSol_\wedge((\Theta, \psi), (\Theta', \psi'), LMap) = (combL(\Theta, LMap) \wedge \Theta', \psi \wedge \psi')$ where

$$\begin{aligned} combL(\Theta, LMap) = &\{P \mapsto \Theta(P) \mid P \notin ran(LMap)\} \\ &\cup \{P \mapsto \textstyle\bigwedge_{LMap(P')=P} \Theta(P') \mid P \in ran(LMap)\} \end{aligned}$$

We show the correctness of $\mathcal{A}_{samp}$ assuming that $\mathcal{A}_{hj}$ works correctly (i.e., it returns a non-empty abstract solution space to the input conjunctive body-disjoint HCCS if it is solvable and otherwise returns *NoSol*).

Theorem 2 (Correctness of $\mathcal{A}_{samp}$). *Given a conjunctive HCCS $(\mathcal{H}, P_\perp)$, $\mathcal{A}_{samp}((\mathcal{H}, P_\perp))$ returns a non-empty abstract solution space of $(\mathcal{H}, P_\perp)$ if $(\mathcal{H}, P_\perp)$ is solvable, and otherwise it returns NoSol.*

We note that it is possible to solve a conjunctive (or non-conjunctive) HCCS more directly by expanding the HCCS to eliminate body-joining and head-joining predicate variables so that it is reduced to a tree-like form [11, 13, 14]. However, the approach makes exponentially many copies of predicate variables whose solutions are conjuncted to form the solution of the original, which often results in complex solutions. $\mathcal{A}_{samp}$ avoids complicating the solution by only making linearly many copies of predicate variables and only copying body-joining predicate variables (assuming that simple solutions are inferred for the root and body-joining predicate variables), and is therefore more likely to infer simple solutions for the whole.[6] In Section 4, we compare our approach with the constraint-expansion approach and show that our approach infers simpler solutions that aid the verification process.[7]

Also, in the implementation, we optimize the solution combination operation $combSol_\wedge$ so that instead of always taking the conjunction of the inferred solutions as described above, we eagerly apply constraint solving to reduce the number of atoms in the combined abstract solution space whenever possible (cf. Example 3).

Example 3. Let $(\mathcal{H}_{ex2}, P_\perp)$ be the HCCS $(Samples, P_\perp)$ given to $\mathcal{A}_{samp}$ in the last iteration of $\mathcal{A}_{solve}$ in Example 2. S is initialized to $(\{P_\perp \mapsto \lambda().\perp\}, \top)$.

[6] Our approach still exponentially expands the constraints, due to $lsol(\cdot)$. It only avoids (always) making exponentially many copies of the predicate variables.

[7] The comparison is with $\mathcal{A}_{solve}$ for solving the whole HCCS and not with $\mathcal{A}_{samp}$ that is just used to solve a sample set.

Because, $\mathit{bjnpvs}(\mathcal{H}_{ex2}) = \{P, Q\}$, $\mathit{initWS}(\mathcal{H}_{ex2}) = \{\mathcal{H}_1, \mathcal{H}_2, \mathcal{H}_3\}$ where

$$\begin{array}{l}\mathcal{H}_1 = \{P_\bot() \Leftarrow Q(x,y) \wedge Q(-x,-y)\}\\ \mathcal{H}_2 = \{Q(x,y) \Leftarrow P(x,y,z) \wedge z = 0, Q(x,y) \Leftarrow P(x,y,z) \wedge z = 2\}\\ \mathcal{H}_3 = \{P(x,y,z) \Leftarrow x \geq z \wedge y \geq 2 - z\}\end{array}$$

$\mathcal{H}_1$ is chosen as the first element to be solved from the workset, and we have $\mathit{MkCnsts}(\mathcal{H}_1, S, \mathcal{H}) = (\{(\mathcal{H}_{ex3}, P'_\bot)\}, \mathit{LMap})$ where $\mathit{LMap} = \{Q_1 \mapsto Q, Q_2 \mapsto Q\}$ and $\mathcal{H}_{ex3}$ is the set of clauses below.

$$\begin{array}{l}\{Q_i(x,y) \Leftarrow x \geq 0 \wedge y \geq 2, Q_i(x,y) \Leftarrow x \geq 2 \wedge y \geq 0 \mid i = 1,2\} \cup\\ \{P_\bot() \Leftarrow Q_1(x,y) \wedge Q_2(-x,-y), P'_\bot \Leftarrow P_\bot() \wedge \neg\bot\}\end{array}$$

$\mathcal{A}_{samp}$ then applies $\mathcal{A}_{hj}$ to $(\mathcal{H}_{ex3}, P'_\bot)$ and obtains an abstract solution space $S_1 = (\Theta_{ex3}, \phi_{ex3})$ (see Example 4 for details) where

$$\begin{array}{rl}\Theta_{ex3} = & \{Q_i \mapsto \lambda(x,y).c_{i,0} + c_{i,1} \cdot x + c_{i,2} \cdot y \geq 0 \mid i = 1,2\} \cup\\ & \{P_\bot \mapsto \lambda().\bot, P'_\bot \mapsto \lambda().\bot\}\\ \phi_{ex3} \equiv & c_{1,0} + c_{2,0} < 0 \wedge c_{1,1} = c_{2,1} \geq 0 \wedge c_{1,2} = c_{2,2} \geq 0 \wedge\\ & \bigwedge_{i=1,2}(c_{i,0} \geq -2 \cdot c_{i,2} \wedge c_{i,0} \geq -2 \cdot c_{i,1})\end{array}$$

$\mathcal{A}_{samp}$ then combines the solution space to update S to $\mathit{combSol}_\wedge(S_1, S, \mathit{LMap}) = (\{P_\bot \mapsto \lambda().\bot, Q \mapsto \lambda(x,y).c_{1,0} + c_{1,1} \cdot x + c_{1,2} \cdot y \geq 0 \wedge c_{2,0} + c_{2,1} \cdot x + c_{2,2} \cdot y \geq 0\}, \phi_{ex3})$. In the implementation, we eagerly apply constraint solving to reduce the number of atoms in the combined solution space. In this example, we check if $\sigma_{uni}(\phi_{ex3})$ is satisfiable where $\sigma_{uni} = \{c_{2,i} \mapsto c_{1,i} \mid i = 0,1,2\}$, and if so updates S to $(\{P_\bot \mapsto \lambda().\bot, Q \mapsto \lambda(x,y).c_{1,0} + c_{1,1} \cdot x + c_{1,2} \cdot y \geq 0\}, \sigma_{uni}(\phi_{ex3}))$ instead. Here, $\sigma_{uni}(\phi_{ex3}) = c_{1,0} < 0 \wedge c_{1,1} \geq 0 \wedge c_{1,2} \geq 0 \wedge c_{1,0} \geq -2 \cdot c_{1,2} \wedge c_{1,0} \geq -2 \cdot c_{1,1}$ which is satisfiable.

Next, $\mathcal{A}_{samp}$ chooses $\mathcal{H}_2$ to solve. It updates the space S so that $S(Q)$ is concrete. For example, $S(Q) = \lambda(x,y).x + y \geq 1$. Then, it solves $\mathcal{H}_2$ and updates S by proceeding similarly to the case for $\mathcal{H}_1$. Lastly, $\mathcal{H}_3$ is solved, and $\mathcal{A}_{samp}$ returns the solution space $(\theta_{ex1}, \top)$ from Example 1. ▲

3.2 The Sub-Algorithm $\mathcal{A}_{hj}$

$\mathcal{A}_{hj}$ takes as input a conjunctive body-disjoint (but possibly head-joined) HCCS. To infer simple solutions to the given HCCS, $\mathcal{A}_{hj}$ first checks if the given HCCS has a solution in the simplest space that maps each predicate variable to a predicate consisting of a single atom, that we call an *atomic solution*, and decomposing the HCCS into smaller subparts containing less head-joining predicate variables if no atomic solution is found. $\mathcal{A}_{hj}$ calls itself recursively to do the decomposition until a solution is found, and the solution spaces of the decomposed subparts are combined to form the solution space for the whole. The key observation here is that, as we shall show in Lemma 4, a conjunctive body-disjoint HCCS with no head-joining predicate variable (i.e., is head-disjoint) is guaranteed to either has an atomic solution or no solution at all. Therefore, the decomposition process is guaranteed to converge to either find a solution or detect that the input HCCS is unsolvable.

```
01: A_hj((H, P⊥)) =
02:   match A_atom((H, P⊥)) with
03:       Sol(S) → return Sol(S)
04:     | NoSol →
05:       if hjnpvs(H) = ∅ then
06:         return NoSol
07:       else
08:         let H1, H2 = Decomp(H) in
09:         match A_hj((H1, P⊥)), A_hj((H2, P⊥)) with
10:             NoSol, _ | _, NoSol → return NoSol
11:           | Sol(S1), Sol(S2) → Sol(combSol∨(S1, S2))
```

Fig. 3. The Sub-Algorithm $\mathcal{A}_{hj}$

Figure 3 shows the overview of $\mathcal{A}_{hj}$. $\mathcal{A}_{hj}$ first calls $\mathcal{A}_{atom}$, whose details are deferred to Section 3.3, to check if there exists an atomic solution to the given HCCS. If so, then it returns the inferred space of atomic solutions (line 3). Otherwise, it checks if the given HCCS is head-disjoint. If so, then there can be no solution to the given HCCS, and we return *NoSol* (line 6). Otherwise, we pick a head-joining predicate variable, say P, and decompose the $\mathcal{H}$ into $\mathcal{H}_1$ and $\mathcal{H}_2$ such that $\mathcal{H}_1$ and $\mathcal{H}_2$ split the clauses in $\mathcal{H}$ whose head is P (along with their "subtree" clauses). (The details of the decomposition is quite intricate and deferred to later in the section.) Then, we call $\mathcal{A}_{hj}$ recursively to infer the solutions for the subparts $\mathcal{H}_1$ and $\mathcal{H}_2$. If either part is found to be unsolvable, then we return *NoSol* (line 10). Otherwise, we combine the returned solution spaces for the subparts to obtain the solution space for the whole (line 11). The combination operation $combSol_\vee$ is analogous to $combSol_\wedge$ used in $\mathcal{A}_{samp}$ except that we take a point-wise disjunction of the solutions as opposed to taking a conjunction (and that there is no management of the copied leaf predicate variables). More formally, $combSol_\vee((\Theta, \psi), (\Theta', \psi')) = (\Theta \vee \Theta', \psi \wedge \psi')$ where $\Theta \vee \Theta'$ is the point-wise disjunction of Θ and Θ'. As with $combSol_\wedge$, in the implementation, we eagerly apply constraint solving to reduce the number of atoms in the combined solution space instead of always taking the disjunction.

We describe the details of the decomposition operation *Decomp*. As remarked above, the role of *Decomp* is to decompose the input HCCS into parts that contain fewer clauses that are head-joined. This is done by selecting some head-joining predicate variable and making two copies of the original that split the portion of the subtrees reachable from the selected predicate variable. More formally, given a non-head-disjoint $\mathcal{H}$, $Decomp(\mathcal{H})$ returns $(\mathcal{H}_1, \mathcal{H}_2)$ as follows. We pick some head-joining predicate variable $P \in hjnpvs(\mathcal{H})$. Let $\mathcal{H}_P$ be the set of clauses in $\mathcal{H}$ having P as the head. We partition $\mathcal{H}_P$ into non-empty disjoint subsets $\mathcal{H}'_1$ and $\mathcal{H}'_2$. For each $\mathcal{H}'_i$ ($i \in \{1, 2\}$), let $\mathcal{H}''_i$ be the set of clauses in $\mathcal{H}$ whose head is Q where $R \triangleleft^*_{\mathcal{H}} Q$ for some predicate variable R that appears in the body of a clause in $\mathcal{H}'_i$. Then, we set $\mathcal{H}_i = \mathcal{H} \setminus (\mathcal{H}'_i \cup \mathcal{H}''_i)$.

We show the correctness of $\mathcal{A}_{hj}$, assuming that the sub-algorithm $\mathcal{A}_{atom}$ works correctly. As we show in Section 3.3, $\mathcal{A}_{atom}$ checks if there exists an atomic solution to the conjunctive body-disjoint (but possibly head-joined) HCCS given as the input, and it is guaranteed to return a non-empty abstract solution space if the given HCCS is solvable and head-disjoint. Then, the following theorem follows from the property of *Decomp* and $combSol_\vee$ and the fact that the recursive

decompositions may happen only as many times as the number of head-joined clauses in the input HCCS.

Theorem 3 (Correctness of $\mathcal{A}_{hj}$). *Given a conjunctive body-disjoint HCCS $(\mathcal{H}, P_\bot)$, $\mathcal{A}_{hj}((\mathcal{H}, P_\bot))$ returns a non-empty abstract solution space of $(\mathcal{H}, P_\bot)$ if $(\mathcal{H}, P_\bot)$ is solvable, and otherwise it returns NoSol.*

The above description of *Decomp* leaves freedom on how to actually do the decomposition, that is, which head-joining predicate variable to select and how to split the subtrees reachable from the selected predicate variable. While Theorem 3 holds true regardless of how the decomposition is done, choosing a coarse decomposition is important for inferring a simple solution. To this end, in the implementation described in Section 4, we choose the decomposition by analyzing the reason for $\mathcal{A}_{atom}$'s failure on finding an atomic solution (cf. line 2) which is returned as an unsatisfiable core of the constraints that $\mathcal{A}_{atom}$ attempted to solve. In addition, instead of doing the recursive decompositions independently for the parts $\mathcal{H}_1$ and $\mathcal{H}_2$ as in Figure 3, we synchronize the decompositions in the recursive call branches to minimize the unnecessary decompositions.[8]

Example 4. Recall $(\mathcal{H}_{ex3}, P'_\bot)$ from Example 3. $(\mathcal{H}_{ex3}, P'_\bot)$ has an atomic solution, and therefore, the first call to $\mathcal{A}_{atom}$ by $\mathcal{A}_{hj}$ (line 3) immediately succeeds and returns an abstract solution space. Here, the returned abstraction solution space is $(\mathcal{H}_{ex3}, P'_\bot)$ from Example 3 (see Example 5 for details).

3.3 The Sub-Algorithm $\mathcal{A}_{atom}$

As remarked in Section 3.2, $\mathcal{A}_{atom}$ decides if there exists an atomic solution to the given conjunctive body-disjoint HCCS, and returns a non-empty abstract solution space of atomic solutions if so. Given the input HCCS $(\mathcal{H}, P_\bot)$, $\mathcal{A}_{atom}$ prepares the *atomic solution template* $\Theta_\mathcal{H}$ that maps each predicate variable $P \in \mathit{pvs}(\mathcal{H})$ to the formula template of the form $\lambda(x_1, \ldots, x_{ar(P)}).c_0 + \Sigma_{i=1}^{ar(P)} c_i \cdot x_i \geq 0$ where c_i's are fresh unknowns.[9] Then, it generates the constraint $\phi_\mathcal{H} = \mathit{constr}(\Theta_\mathcal{H}, (\mathcal{H}, P_\bot)) = c < 0 \wedge \bigwedge_{hc \in \mathcal{H}} \mathit{constr}(\Theta_\mathcal{H}, hc)$ where $\Theta_\mathcal{H}(P_\bot) = \lambda().c \geq 0$ and $\mathit{constr}(\Theta_\mathcal{H}, hc)$ is defined as follows; for $hc = a_0 \Leftarrow \bigwedge_{i=1}^{\ell} a_i \wedge \bigwedge_{i=1}^{q} p_i$ with $\mathit{fvs}(hc) = \{x_1, \ldots, x_m\}$,

$$\mathit{constr}(\Theta_\mathcal{H}, hc) = \textstyle\bigwedge_{i=1}^{q} \alpha_i \geq 0 \wedge \bigwedge_{j=0}^{m} t_{0,j} = (\sum_{i=1}^{\ell} t_{i,j}) + (\sum_{i=1}^{q} \alpha_i \cdot r_{i,j})$$

where $\alpha_1, \ldots, \alpha_q$ are fresh unknowns, and $\Theta_\mathcal{H}(a_i)$ and p_i are respectively $t_{i,0} + \sum_{j=1}^{m} t_{i,j} \cdot x_j \geq 0$ (for $i \in \{0, \ldots, \ell\}$) and $r_{i,0} + \sum_{j=1}^{m} r_{i,j} \cdot x_j \geq 0$ (for $i \in \{1, \ldots, q\}$) for some linear terms over unknowns $t_{i,j}$ and rational constants $r_{i,j}$. Note that $\phi_\mathcal{H}$ is a QFLRA formula over unknowns (i.e., it contains no variable or product of unknowns).

[8] *Decomp* is similar in spirit to the sample set "split" operation from [1].

[9] For simplicity, in this section, we only consider non-strict inequality atoms. Strict inequalities can be handled similarly by using the Motzkin's transposition theorem instead of the Farkas' lemma (cf. the extended report [21]).

Then, $\mathcal{A}_{atom}$ checks if $\phi_{\mathcal{H}}$ is satisfiable, that is, if there exists an assignment σ to the unknowns such that $\models \sigma(\phi_{\mathcal{H}})$, and if so, returns $(\Theta_{\mathcal{H}}, \phi_{\mathcal{H}})$ as the abstract solution space. Otherwise, it detects that $(\mathcal{H}, P_\bot)$ has no atomic solution and returns *NoSol*. We state and prove the correctness of $\mathcal{A}_{atom}$.

Theorem 4 (Correctness of $\mathcal{A}_{atom}$). *Given a conjunctive body-disjoint HCCS $(\mathcal{H}, P_\bot)$, $\mathcal{A}_{atom}((\mathcal{H}, P_\bot))$ returns a non-empty abstract atomic solution space S of $(\mathcal{H}, P_\bot)$ if $(\mathcal{H}, P_\bot)$ has an atomic solution, and otherwise returns NoSol.*

Also, the following holds by the Farkas' lemma [16].

Lemma 4. *A conjunctive body-disjoint and head-disjoint HCCS either has an atomic solution or no solution.*

Therefore, $\mathcal{A}_{atom}$ completely decides the solvability of a conjunctive body-disjoint head-disjoint HCCS. In general, a solvable conjunctive body-disjoint (but not head-disjoint) HCCS may not be atomically solvable. For example, $(\mathcal{H}, P_\bot)$ where $\mathcal{H} = \{P(x,y) \Leftarrow x \leq 0 \wedge y \leq 1, P(x,y) \Leftarrow x \leq 1 \wedge y \leq 0, P_\bot() \Leftarrow P(x,y) \wedge x > 0 \wedge y > 0\}$ is solvable but has no atomic solution. Thus, when $\mathcal{A}_{atom}$ fails to find an atomic solution to such an HCCS, the information is propagated back to $\mathcal{A}_{hj}$ to decompose some head-joined clauses.

Example 5. Consider the HCCS $(\mathcal{H}_{ex3}, P'_\bot)$ from Example 3 (note that it is head-disjoint). $\mathcal{A}_{atom}$ prepares the atomic solution template $\Theta_{\mathcal{H}_{ex3}} = \{Q_i \mapsto \lambda(x,y).c_{i,0} + c_{i,1} \cdot x + c_{i,2} \cdot y \geq 0 \mid i = 1,2\} \cup \{P_\bot \mapsto \lambda().c_{3,0} \geq 0, P'_\bot \mapsto \lambda().c_{4,0} \geq 0\}$ and generates the constraint $constr(\Theta_{\mathcal{H}_{ex3}}, (\mathcal{H}_{ex3}, P'_\bot))$:

$$
\begin{array}{l}
c_{4,0} < 0 \wedge \alpha_1, \alpha_2 \geq 0 \wedge c_{4,0} = c_{3,0} + \alpha_1 \wedge c_{3,0} = c_{1,0} + c_{2,0} + \alpha_2 \wedge \\
0 = c_{1,1} - c_{2,1} \wedge 0 = c_{1,2} - c_{2,2} \wedge \\
\bigwedge_{i=1,2} \left(\begin{array}{l} \alpha_{i,1}, \alpha_{i,2}, \alpha_{i,3} \geq 0 \wedge c_{i,0} = -2 \cdot \alpha_{i,2} + \alpha_{i,3} \wedge c_{i,1} = \alpha_{i,1} \wedge c_{i,2} = \alpha_{i,2} \wedge \\ \alpha_{i,4}, \alpha_{i,5}, \alpha_{i,6} \geq 0 \wedge c_{i,0} = -2 \cdot \alpha_{i,4} + \alpha_{i,6} \wedge c_{i,1} = \alpha_{i,4} \wedge c_{i,2} = \alpha_{i,5} \end{array} \right)
\end{array}
$$

In the constraint generation, we add the tautology $1 \geq 0$ to the body of each clause. This often widens the obtained solution space. After satisfiability checking and simplification, $\mathcal{A}_{atom}$ returns the abstract solution space $(\Theta_{ex3}, \phi_{ex3})$ given in Example 3. ▲

4 Implementation and Experiments

We have implemented a prototype of the new constraint solving algorithm $\mathcal{A}_{solve}$. We use the linear programming tool GLPK (`http://www.gnu.org/software/glpk`) for the linear constraint solving that is used to operate on abstract solution spaces, and Z3 (`http://z3.codeplex.com`) for the unsat core generation in $\mathcal{A}_{hj}$ and for checking the candidate solution against the whole HCCS in $\mathcal{A}_{solve}$. We use the objective function in linear programming to find a model with small valuations to further bias towards simple solutions.

We use the constraint solver as the backend of the MoCHi software model checker [9]. MoCHi verifies assertion safety of OCaml programs via predicate abstraction, higher-order model checking, and CEGAR. MoCHi is a good platform

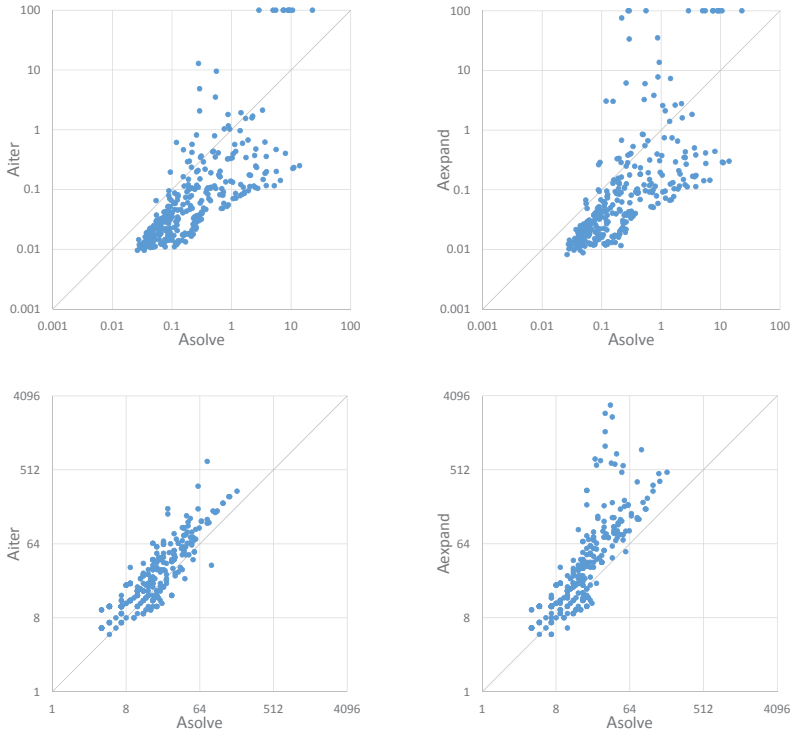

Fig. 4. Run time (upper) and solution size (lower) comparisons of the HCCS solving algorithms on benchmarks HCCSs

for experimenting with the constraint solver because the Horn-clause constraints solved there often have a complex structure. (Intuitively, this is because the constraints express the flow of data in the program to be verified, and data often flow in a complex way in a functional program, e.g., passed to and returned from recursive functions, captured in closures, etc.)

We compare the new algorithm $\mathcal{A}_{solve}$ with two other algorithms, $\mathcal{A}_{iter}$ and $\mathcal{A}_{expand}$. $\mathcal{A}_{iter}$ is an implementation of the iterative approach to solving HCCS [20, 18], and is also used in the previous work on MoCHi [9, 15, 22, 10]. $\mathcal{A}_{expand}$ is an implementation of the constraint-expansion approach [4, 14] in which the given HCCS is first expanded into a body-disjoint head-disjoint HCCS and the iterative algorithm is used to solve the resulting HCCS. (See also **Related Work** in Section 1.)

We have ran the three algorithms on 327 HCCSs generated by running MoCHi with $\mathcal{A}_{solve}$ on 139 benchmark programs, most of which are taken from the previous work on MoCHi [9, 15, 22, 10]. We measured the time spent on solving each HCCS by each algorithm as well as the size of the inferred solution (the sum of the syntactic sizes of the predicates). We also compare the overall

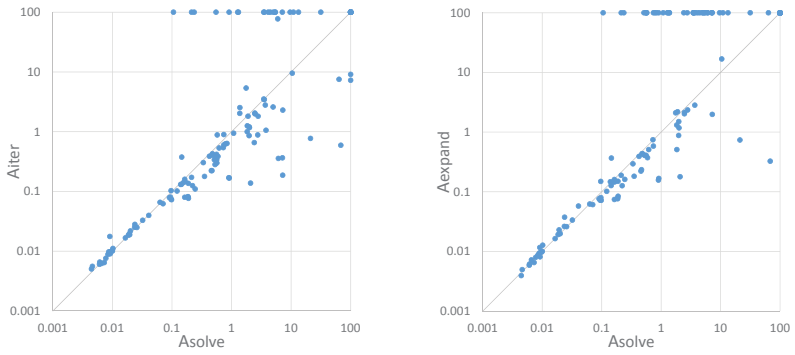

Fig. 5. Run time comparison of the HCCS solving algorithms on benchmarks programs

verification speed of MoCHi when using the three algorithms on the 139 benchmark programs. The experiments were conducted on a machine with 2.69 GHz i7-4600U processor with 16 GB of RAM, with the time limit of 100 seconds. The benchmark programs, the benchmark HCCSs and the experiment results are available online [21].

Figure 4 shows the scatter plots that compare the run times and the solution sizes of $\mathcal{A}_{solve}$, $\mathcal{A}_{iter}$, and $\mathcal{A}_{expand}$ on each of the 318 benchmark HCCSs. The run time plots show that, on most instances, $\mathcal{A}_{solve}$ is slower than $\mathcal{A}_{iter}$ and $\mathcal{A}_{expand}$ due to the additional effort to find a simple solution. The plots also show that $\mathcal{A}_{solve}$ is sometimes faster than the other two. The behavior is attributed to the fact that $\mathcal{A}_{solve}$ is sometimes able to find a solution for the whole by sampling a very small fraction of the given HCCS, and the fact that $\mathcal{A}_{iter}$ and $\mathcal{A}_{expand}$ (after the expansion) uses the iterative approach which can be sometimes slow on large instances. The solution size plots show that $\mathcal{A}_{solve}$ is able to compute smaller solutions than the other two on most instances.

Figure 5 shows the plots comparing the run times of the overall verification process on each of the 139 benchmark programs for each constraint solving algorithm. The plots show that, with the new algorithm $\mathcal{A}_{solve}$, MoCHi is able to verify significantly more programs within the time limit than with the other two algorithms. The plots also show that the heavier cost of constraint solving in the new algorithm is often compensated by the better predicates inferred, thereby allowing the overall verification speed to match those of the other algorithms even on instances that the other algorithms were able to verify in time.

5 Conclusion

We have presented a new approach to solving recursion-free Horn-clause constraints. Our approach is inspired by the sampling-based approach to inferring simple interpolants [17, 1] and is geared toward inferring simple solutions. We

have shown that the new approach is effective at inferring simple solutions that are useful to program verification.

References

1. A. Albarghouthi and K. L. McMillan. Beautiful interpolants. In *CAV*, pages 313–329, 2013.
2. N. Bjørner, K. L. McMillan, and A. Rybalchenko. On solving universally quantified horn clauses. In *SAS*, pages 105–125, 2013.
3. W. Craig. Linear reasoning. a new form of the herbrand-gentzen theorem. *The Journal of Symbolic Logic*, 22(03):250–268, 1957.
4. S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416, 2012.
5. A. Gupta, C. Popeea, and A. Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In *POPL*, pages 331–344, 2011.
6. A. Gupta, C. Popeea, and A. Rybalchenko. Solving recursion-free horn clauses over LI+UIF. In *APLAS*, pages 188–203, 2011.
7. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244, 2004.
8. K. Hoder, L. Kovács, and A. Voronkov. Playing in the grey area of proofs. In *POPL*, pages 259–272, 2012.
9. N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and CEGAR for higher-order model checking. In *PLDI*, pages 222–233. ACM, 2011.
10. T. Kuwahara, T. Terauchi, H. Unno, and N. Kobayashi. Automatic termination verification for higher-order functional programs. In *ESOP*, pages 392–411, 2014.
11. K. McMillan and A. Rybalchenko. Computing relational fixed points using interpolation. Technical Report MSR-TR-2013-6, January 2013.
12. K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, pages 123–136, 2006.
13. P. Rümmer, H. Hojjat, and V. Kuncak. Classifying and solving horn clauses for verification. In *VSTTE*, pages 1–21, 2013.
14. P. Rümmer, H. Hojjat, and V. Kuncak. Disjunctive interpolants for horn-clause verification. In *CAV*, pages 347–363, 2013.
15. R. Sato, H. Unno, and N. Kobayashi. Towards a scalable software model checker for higher-order programs. In *PEPM*, pages 53–62. ACM, 2013.
16. A. Schrijver. *Theory of linear and integer programming*. Wiley, 1998.
17. R. Sharma, A. V. Nori, and A. Aiken. Interpolants as classifiers. In *CAV*, pages 71–87, 2012.
18. T. Terauchi. Dependent types from counterexamples. In *POPL*, pages 119–130, 2010.
19. T. Terauchi and H. Unno. Relaxed stratification: A new approach to practical complete predicate refinement. In *ESOP*, 2015. To appear.
20. H. Unno and N. Kobayashi. Dependent type inference with interpolants. In *PPDP*, pages 277–288, 2009.
21. H. Unno and T. Terauchi. Inferring simple solutions to recursion-free horn clauses via sampling. 2015. `http://www.cs.tsukuba.ac.jp/~uhiro`.
22. H. Unno, T. Terauchi, and N. Kobayashi. Automating relatively complete verification of higher-order functional programs. In *POPL*, pages 75–86, 2013.