Automating Induction for Solving Horn Clauses

Hiroshi Unno, Sho Torii, and Hiroki Sakamoto

University of Tsukuba
{uhiro,sho,sakamoto}@logic.cs.tsukuba.ac.jp

Abstract. Verification problems of programs in various paradigms can be reduced to problems of solving Horn clause constraints on predicate variables that represent unknown inductive invariants. This paper presents a novel Horn constraint solving method based on inductive theorem proving: the method reduces Horn constraint solving to validity checking of first-order formulas with inductively defined predicates, which are then checked by induction on the derivation of the predicates. To automate inductive proofs, we introduce a novel proof system tailored to Horn constraint solving, and use a PDR-based Horn constraint solver as well as an SMT solver to discharge proof obligations arising in the proof search. We prove that our proof system satisfies the soundness and relative completeness with respect to ordinary Horn constraint solving schemes. The two main advantages of the proposed method are that (1) it can deal with constraints over any background theories supported by the underlying SMT solver, including nonlinear arithmetic and algebraic data structures, and (2) the method can verify *relational specifications* across programs in various paradigms where multiple function calls need to be analyzed simultaneously. The class of specifications includes practically important ones such as functional equivalence, associativity, commutativity, distributivity, monotonicity, idempotency, and non-interference. Our novel combination of Horn clause constraints with inductive theorem proving enables us to naturally and automatically axiomatize recursive functions that are possibly non-terminating, nondeterministic, higher-order, exception-raising, and over non-inductively defined data types. We have implemented a relational verification tool for the OCaml functional language based on the proposed method and obtained promising results in preliminary experiments.

1 Introduction

Verification problems of programs written in various paradigms, including imperative [30], logic, concurrent [28], functional [47, 54, 55, 59], and object-oriented [36] ones, can be reduced to problems of solving Horn clause constraints on predicate variables that represent unknown inductive invariants. A given program is guaranteed to satisfy its specification if the Horn constraints generated from the program have a solution (see [27] for an overview of the approach).

This paper presents a novel Horn constraint solving method based on inductive theorem proving: the method reduces Horn constraint solving to validity

checking of first-order formulas with inductively defined predicates, which are then checked by induction on the derivation of the predicates. The main technical challenge here is how to automate inductive proofs. To this end, we propose an inductive proof system tailored for Horn constraint solving and a technique based on SMT and PDR [10] to automate proof search in the system. Furthermore, we prove that the proof system satisfies the soundness and relative completeness with respect to ordinary Horn constraint solving schemes.

Compared to previous Horn constraint solving methods [27, 29, 32, 33, 41, 48, 52, 55, 57] based on Craig interpolation [21, 42], abstract interpretation [20], and PDR, the proposed method has two major advantages:

1. It can solve Horn clause constraints over any background theories supported by the underlying SMT solver. Our method solved constraints over the theories of nonlinear arithmetic and algebraic data structures, which are not supported by most existing Horn constraint solvers.
2. It can verify *relational specifications* where multiple function calls need to be analyzed simultaneously. The class of specifications includes practically important ones such as functional equivalence, associativity, commutativity, distributivity, monotonicity, idempotency, and non-interference.

To show the usefulness of our approach, we have implemented a relational verification tool for the OCaml functional language based on the proposed method and obtained promising results in preliminary experiments.

For an example of the reduction from (relational) verification to Horn constraint solving, consider the following OCaml program D_{mult}.[1]

```
let rec mult x y = if y=0 then 0 else x + mult x (y-1)
let rec mult_acc x y a = if y=0 then a else mult_acc x (y-1) (a+x)
let main x y a = assert (mult x y + a = mult_acc x y a)
```

Here, the function `mult` takes two integer arguments `x`, `y` and recursively computes `x` $\times$ `y` (note that `mult` never terminates if `y` < 0). `mult_acc` is a tail-recursive version of `mult` with an accumulator `a`. The function `main` contains an assertion with the condition `mult x y + a = mult_acc x y a`, which represents a relational specification, namely, the functional equivalence of `mult` and `mult_acc`. Our verification problem here is whether for any integers x, y, and a, the evaluation of `main` x y a, under the call-by-value evaluation strategy adopted by OCaml, never causes an assertion failure, that is $\forall x, y, a \in \mathbb{N}$. `main` x y $a \not\rightarrow^*$`assert false`. By using a constraint generation method for functional programs [55], the relational verification problem is reduced to the constraint solving problem of the following Horn clause constraint set $\mathcal{H}_{mult}$:

$$\left\{ \begin{array}{l} P(x,0,0), \quad P(x,y,x+r) \Leftarrow P(x,y-1,r) \wedge (y \neq 0), \\ Q(x,0,a,a), \quad Q(x,y,a,r) \Leftarrow Q(x,y-1,a+x,r) \wedge (y \neq 0), \\ \bot \Leftarrow P(x,y,r_1) \wedge Q(x,y,a,r_2) \wedge (r_1 + a \neq r_2) \end{array} \right\}$$

[1] Our work also applies to programs that require a path-sensitive analysis of intricate control flows caused by non-termination, non-determinism, higher-order functions, and exceptions but, for illustration purposes, we use this as a running example.

Here, the predicate variable P (resp. Q) represents an inductive invariant among the arguments and the return value of the function `mult` (resp. `mult_acc`). The first Horn clause $P(x, 0, 0)$ is generated from the then-branch of the definition of `mult` and expresses that `mult` returns 0 if 0 is given as the second argument. The second clause in $\mathcal{H}_{mult}$, $P(x, y, x+r) \Leftarrow P(x, y-1, r) \wedge (y \neq 0)$ is generated from the else-branch and represents that `mult` returns $x+r$ if the second argument y is non-zero and r is returned by the recursive call `mult x (y-1)`. The other Horn clauses are similarly generated from the then- and else- branches of `mult_acc` and the assertion in `main`. Because $\mathcal{H}_{mult}$ has a satisfying substitution (i.e., solution) $\theta_{mult} = \{P \mapsto \lambda(x, y, r).x \times y = r, Q \mapsto \lambda(x, y, a, r).x \times y + a = r\}$ for the predicate variables P and Q, the correctness of the constraint generation [55] guarantees that the evaluation of `main` x y a never causes an assertion failure.

The previous Horn constraint solving methods, however, cannot solve this kind of constraints that require a relational analysis of multiple predicates. To see why, recall the constraint in $\mathcal{H}_{mult}$, $\bot \Leftarrow P(x, y, r_1) \wedge Q(x, y, a, r_2) \wedge (r_1 + a \neq r_2)$ which asserts the equivalence of `mult` and `mult_acc`, where a relational analysis of the two predicates P and Q is required. The previous methods, however, analyze each predicate P and Q separately, and therefore must infer nonlinear invariants $r_1 = x \times y$ and $r_2 = x \times y + a$ respectively for the predicate applications $P(x, y, r_1)$ and $Q(x, y, a, r_2)$ to conclude $r_1 + a = r_2$ by canceling $x \times y$, because x and y are the only shared arguments between $P(x, y, r_1)$ and $Q(x, y, a, r_2)$. The previous methods can only find solutions that are expressible by efficiently decidable theories such as the quantifier-free linear real (QF_LRA) and integer (QF_LIA) arithmetic[2], which are not powerful enough to express the above nonlinear invariants and the solution θ_{mult} of $\mathcal{H}_{mult}$.

By contrast, our induction-based Horn constraint solving method can directly and automatically show that the predicate applications $P(x, y, r_1)$ and $Q(x, y, a, r_2)$ imply $r_1 + a = r_2$ (i.e., $\mathcal{H}_{mult}$ is solvable), by simultaneously analyzing the two. More precisely, our method interprets P, Q as the predicates inductively defined by the definite clauses (i.e., the clauses whose head is a predicate application), and uses induction on the derivation of $P(x, y, r_1)$ to prove the conjecture $\forall x, y, r_1, a, r_2.(P(x, y, r_1) \wedge Q(x, y, a, r_2) \wedge (r_1 + a \neq r_2) \Rightarrow \bot)$ denoted by the goal clause (i.e., the clause whose head is *not* a predicate application).

The use of Horn clause constraints, which can be considered as an Intermediate Verification Language (IVL) common to Horn constraint solvers and target languages, enables our method to verify relational specifications across programs written in various paradigms. Horn constraints can naturally axiomatize various advanced language features including recursive functions that are partial (i.e., possibly non-terminating), non-deterministic, higher-order, exception-raising, and over non-inductively defined data types (recall that $\mathcal{H}_{mult}$ axiomatizes the partial functions `mult` and `mult_acc`, and see the full version [58] for more examples). Furthermore, we can automate the axiomatization process by using program logics such as Hoare logics for imperative and refinement type systems [47, 54, 55, 60] for functional programs. In fact, researchers have developed

[2] See `http://smt-lib.org/` for the definition of the theories.

and made available tools such as SeaHorn [30] and JayHorn [36], respectively for translating C and Java programs into Horn constraints. Despite their expressiveness, Horn constraints have a simpler logical semantics than other popular IVLs like Boogie [3] and Why3 [8]. The simplicity enabled us to directly apply inductive theorem proving and made the proofs and implementation easier.

In contrast to our method based on the logic of predicates defined by Horn clause constraints, most state-of-the-art automated inductive theorem provers such as ACL2s [15], Leon [50], Dafny [40], Zeno [49], HipSpec [18], and CVC4 [46] are based on logics of pure total functions over inductively-defined data structures. Some of them support powerful induction schemes such as recursion induction [43] and well-founded induction (if the termination arguments for the recursive functions are given). However, the axiomatization process often requires users' manual intervention and possibly has a negative effect on the automation of induction later, because one needs to take into consideration the evaluation strategies and complex control flows caused by higher-order functions and side-effects such as non-termination, exceptions, and non-determinism. Furthermore, the process needs to preserve branching and calling context information for path- and context-sensitive verification. Thus, our approach complements automated inductive theorem proving with the expressive power of Horn clause constraints and, from the opposite point of view, opens the way to leveraging the achievements of the automated induction community into Horn constraint solving.

The rest of the paper is organized as follows. In Section 2, we will give an overview of our induction-based Horn constraint solving method. Section 3 defines Horn constraint solving problems and proves the correctness of the reduction from constraint solving to inductive theorem proving. Section 4 formalizes our constraint solving method and proves its soundness and relative completeness. Section 5 reports on our prototype implementation based on the proposed method and the results of preliminary experiments. We compare our method with related work in Section 6 and conclude the paper with some remarks on future work in Section 7. The full version [58] contains omitted proofs, example constraints generated from verification problems, and implementation details.

2 Overview of Induction-Based Horn Constraint Solving

In this section, we use the constraint set $\mathcal{H}_{mult}$ in Section 1 as a running example to give an overview of our induction-based Horn constraint solving method (more formal treatment is provided in Sections 3 and 4). Our method interprets the definite clauses of a given constraint set as derivation rules for *atoms* $P(\widetilde{t})$, namely, applications of a predicate variable P to a sequence $\widetilde{t}$ of terms $t_1, \ldots, t_m$.

For example, the definite clauses $\mathcal{D}_{mult} \subseteq \mathcal{H}_{mult}$ are interpreted as the rules:

$$\frac{\models y = 0 \wedge r = 0}{P(x, y, r)} \qquad \frac{P(x, y-1, r-x) \qquad \models y \neq 0}{P(x, y, r)}$$

$$\frac{\models y = 0 \wedge r = a}{Q(x, y, a, r)} \qquad \frac{Q(x, y-1, a+x, r) \qquad \models y \neq 0}{Q(x, y, a, r)}$$

Here, the heads of the clauses are changed into the uniform representations $P(x, y, r)$ and $Q(x, y, a, r)$ of atoms over variables. These rules inductively define the least interpretation ρ_{mult} for P and Q that satisfies the definite clauses $\mathcal{D}_{mult}$. We thus get $\rho_{mult} = \{P \mapsto \{(x, y, r) \in \mathbb{Z}^3 \mid x \times y = r \wedge y \geq 0\}, Q \mapsto \{(x, y, a, r) \in \mathbb{Z}^4 \mid x \times y + a = r \wedge y \geq 0\}\}$, and $\mathcal{H}_{mult}$ has a solution iff the goal clause

$$\forall x, y, r_1, a, r_2.\, (P(x, y, r_1) \wedge Q(x, y, a, r_2) \wedge (r_1 + a \neq r_2) \Rightarrow \bot)$$

is valid under ρ_{mult} (see Section 3 for a correctness proof of the reduction). We then check the validity of the goal by induction on the derivation of atoms.

Principle 1 (Induction on Derivations) *Let $\mathcal{P}$ be a property on derivations D of atoms. We then have $\forall D.\mathcal{P}(D)$ if and only if $\forall D.\, ((\forall D' \prec D.\mathcal{P}(D')) \Rightarrow \mathcal{P}(D))$, where $D' \prec D$ represents that D' is a strict sub-derivation of D.*

Formally, we propose an inductive proof system for deriving judgments of the form $\mathcal{D}; \Gamma; A; \phi \vdash \bot$, where $\bot$ represents the contradiction, ϕ represents a formula without atoms, A represents a set of atoms, Γ represents a set of induction hypotheses and user-specified lemmas, and $\mathcal{D}$ represents a set of definite clauses that define the least interpretation of the predicate variables in Γ or A. Here, Γ, A, and ϕ are allowed to have common free term variables. The free term variables of a clause in $\mathcal{D}$ have the scope within the clause, and are considered to be universally quantified. Intuitively, a judgment $\mathcal{D}; \Gamma; A; \phi \vdash \bot$ means that the formula $\bigwedge \Gamma \wedge \bigwedge A \wedge \phi \Rightarrow \bot$ is valid under the least interpretation induced by $\mathcal{D}$. For example, consider the following judgment J_{mult}:

$$J_{mult} \triangleq \mathcal{D}_{mult}; \emptyset; \{P(x, y, r_1), Q(x, y, a, r_2)\}\,; (r_1 + a \neq r_2) \vdash \bot$$

If J_{mult} is derivable, $P(x, y, r_1) \wedge Q(x, y, a, r_2) \wedge (r_1 + a \neq r_2) \Rightarrow \bot$ is valid under the least predicate interpretation by $\mathcal{D}_{mult}$, and hence $\mathcal{H}_{mult}$ has a solution.

The inference rules for the judgment $\mathcal{D}; \Gamma; A; \phi \vdash \bot$ are shown in Section 4, Figure 2. The rules there, however, are too general and formal for the purpose of providing an overview of the idea. Therefore, we defer a detailed explanation of the rules to Section 4, and here explain a simplified version shown below, obtained from the complete version by eliding some conditions and subtleties while retaining the essence. The rules are designed to exploit Γ and $\mathcal{D}$ for iteratively updating the current *knowledge* represented by the formula $\bigwedge A \wedge \phi$ until a contradiction is implied. The first rule INDUCT

$$\frac{\begin{array}{c} P(\widetilde{t}) \in A \qquad \{\widetilde{y}\} = \mathit{fvs}(A) \cup \mathit{fvs}(\phi) \qquad \widetilde{x} : \text{fresh} \qquad \sigma = \{\widetilde{y} \mapsto \widetilde{x}\} \\ \psi = \forall \widetilde{x}.\, \big(\big(P(\sigma\widetilde{t}) \prec P(\widetilde{t})\big) \wedge \bigwedge \sigma A \Rightarrow \neg(\sigma\phi)\big) \qquad \mathcal{D}; \Gamma \cup \{\psi\}\,; A; \phi \vdash \bot \end{array}}{\mathcal{D}; \Gamma; A; \phi \vdash \bot}$$

selects an atom $P(\widetilde{t}) \in A$ and performs induction on the derivation of the atom by adding a new induction hypothesis ψ to Γ, which is obtained from the current proof obligation $\bigwedge A \wedge \phi \Rightarrow \bot$ by generalizing its free term variables (denoted by $\mathit{fvs}(A) \cup \mathit{fvs}(\phi)$) into fresh ones $\widetilde{x}$ using a map σ, and adding a guard $P(\sigma\widetilde{t}) \prec$

$P(\widetilde{t})$, requiring the derivation of $P(\sigma\widetilde{t})$ to be a strict sub-derivation of that of $P(\widetilde{t})$, to avoid an unsound application of ψ. The second rule UNFOLD

$$\frac{P(\widetilde{t}) \in A \qquad \mathcal{D}; \Gamma; A \cup [\widetilde{t}/\widetilde{x}]A'; \phi \wedge [\widetilde{t}/\widetilde{x}]\phi' \vdash \bot \qquad (\text{for each } (P(\widetilde{x}) \Leftarrow A' \wedge \phi') \in \mathcal{D})}{\mathcal{D}; \Gamma; A; \phi \vdash \bot}$$

selects an atom $P(\widetilde{t}) \in A$, performs a case analysis on the last rule used to derive the atom, which is represented by a definite clause in $\mathcal{D}$. The third rule APPLY$\bot$

$$\frac{\begin{array}{c}\forall\widetilde{x}.\left(\left(P(\widetilde{t'}) \prec P(\widetilde{t})\right) \wedge \bigwedge A' \Rightarrow \phi'\right) \in \Gamma \qquad \mathrm{dom}(\sigma) = \{\widetilde{x}\} \qquad P(\sigma\widetilde{t'}) \prec P(\widetilde{t}) \\ \models \bigwedge A \wedge \phi \Rightarrow \bigwedge \sigma A' \qquad \mathcal{D}; \Gamma; A; \phi \wedge \sigma\phi' \vdash \bot\end{array}}{\mathcal{D}; \Gamma; A; \phi \vdash \bot}$$

selects an induction hypothesis in Γ, and tries to find an instantiation σ of the quantified variables $\widetilde{x}$ such that

- the instantiated premise $\bigwedge \sigma A'$ of the hypothesis is implied by the current knowledge $\bigwedge A \wedge \phi$ and
- the derivation of the atom $P(\sigma\widetilde{t'}) \in \sigma A'$ to which the hypothesis is being applied is a strict sub-derivation of that of the atom $P(\widetilde{t})$ on which the induction (that has introduced the hypothesis) has been performed.

If such a σ is found, $\sigma\phi'$ is added to the current knowledge. The fourth rule VALID$\bot$ checks whether the current knowledge implies $\bot$, and if so, closes the proof branch under consideration.

Figure 1 shows the structure (with side-conditions omitted) of a derivation of the judgment J_{mult}, constructed by using the simplified version of the inference rules. We below explain how the derivation is constructed. First, by performing induction on the atom $P(x, y, r_1)$ in J_{mult} using the rule INDUCT, we obtain the subgoal J_0, where the induction hypothesis $\forall x', y', r_1', a', r_2'.\ \phi_{ind}$ is added. We then apply UNFOLD to perform a case analysis on the last rule used to derive the atom $P(x, y, r_1)$, and obtain the two subgoals J_1 and J_2. We here got two subgoals because D_{mult} has two clauses with the head that matches with the atom $P(x, y, r_1)$. The two subgoals are then discharged as follows.

- **Subgoal 1:** By performing a case analysis on $Q(x, y, a, r_2)$ in J_1 using the rule UNFOLD, we further get two subgoals J_3 and J_4. Both J_3 and J_4 are derived by the rule VALID$\bot$ because $\models \phi_3 \Rightarrow \bot$ and $\models \phi_4 \Rightarrow \bot$ hold.
- **Subgoal 2:** By performing a case analysis on $Q(x, y, a, r_2)$ in J_2 using the rule UNFOLD, we obtain two subgoals J_5 and J_6. J_5 is derived by the rule VALID$\bot$ because $\models \phi_5 \Rightarrow \bot$ holds. To derive J_6, we use the rule APPLY$\bot$ to apply the induction hypothesis to the atom $P(x, y-1, r_1-x) \in A_6$ in J_6. Note that this can be done by using the quantifier instantiation

$$\sigma = \{x' \mapsto x, y' \mapsto y-1, r_1' \mapsto r_1 - x, a' \mapsto a + x, r_2' \mapsto r_2\},$$

$$\dfrac{\dfrac{\dfrac{}{J_3}\ (\textsc{Valid}\bot) \quad \dfrac{}{J_4}\ (\textsc{Valid}\bot)}{J_1}\ (\textsc{Unfold}) \quad \dfrac{\dfrac{}{J_5}\ (\textsc{Valid}\bot) \quad \dfrac{\dfrac{}{J_7}\ (\textsc{Valid}\bot)}{J_6}\ (\textsc{Apply}\bot)}{J_2}\ (\textsc{Unfold})}{\dfrac{J_0}{J_{mult}}\ (\textsc{Induct})}$$

Here, J_i's are of the form $J_i \triangleq \mathcal{D}_{mult}; \{\forall x', y', r_1', a', r_2'.\ \phi_{ind}\}; A_i; \phi_i \vdash \bot$ where:

$$\phi_{ind} = (P(x', y', r_1') \prec P(x, y, r_1)) \wedge P(x', y', r_1') \wedge Q(x', y', a', r_2') \Rightarrow r_1' + a' = r_2'$$

$$\begin{array}{ll}
\phi_0 = r_1 + a \neq r_2 & \phi_6 = \phi_2 \wedge y \neq 0 \\
\phi_1 = \phi_0 \wedge y = 0 \wedge r_1 = 0 & \phi_7 = \phi_6 \wedge \sigma(r_1' + a' = r_2') \\
\phi_2 = \phi_0 \wedge y \neq 0 & A_0 = A_1 = A_3 = \{P(x, y, r_1), Q(x, y, a, r_2)\} \\
\phi_3 = \phi_1 \wedge y = 0 \wedge r_2 = a & A_2 = A_5 = A_0 \cup \{P(x, y-1, r_1 - x)\} \\
\phi_4 = \phi_1 \wedge y \neq 0 & A_4 = A_1 \cup \{Q(x, y-1, a + x, r_2)\} \\
\phi_5 = \phi_2 \wedge y = 0 \wedge r_2 = a & A_6 = A_7 = A_2 \cup \{Q(x, y-1, a + x, r_2)\}
\end{array}$$

Fig. 1. The structure of an example derivation of J_{mult}.

because $\sigma(P(x', y', r_1')) = P(x, y-1, r_1 - x) \prec P(x, y, r_1)$ holds and the premise $\sigma(P(x', y', r_1') \wedge Q(x', y', a', r_2')) = P(x, y-1, r_1 - x) \wedge Q(x, y-1, a + x, r_2)$ of the instantiated hypothesis is implied by the current knowledge $\bigwedge A_6 \wedge r_1 + a \neq r_2 \wedge y \neq 0$. We thus obtain the subgoal J_7, whose ϕ-part is equivalent to $r_1 + a \neq r_2 \wedge y \neq 0 \wedge r_1 + a = r_2$. Because this implies a contradiction, J_7 is finally derived by using the rule Valid$\bot$.

To automate proof search in the system, we use either an off-the-shelf SMT solver or a PDR-based Horn constraint solver for checking whether the current knowledge implies a contradiction (in the rule Valid$\bot$). An SMT solver is also used to check whether each element of Γ can be used to update the current knowledge, by finding a quantifier instantiation σ (in the rule Apply$\bot$). The use of an SMT solver provides our method with efficient and powerful reasoning about data structures, including integers, real numbers, arrays, algebraic data types, and uninterpreted functions. However, there still remain two challenges to be addressed towards full automation:

1. **Challenge:** How to check (in the rule Apply$\bot$) the strict sub-derivation relation $P(\widetilde{t'}) \prec P(\widetilde{t})$ between the derivation of an atom $P(\widetilde{t'})$ to which an induction hypothesis in Γ is being applied, and the derivation of the atom $P(\widetilde{t})$ on which the induction has been performed? Recall that in the above derivation of J_{mult}, we needed to check $P(x, y-1, r_1 - x) \prec P(x, y, r_1)$ before applying the rule Apply$\bot$ to J_6.
   Our solution: The formalized rules presented in Section 4 keep sufficient information for checking the strict sub-derivation relation: we associate each induction hypothesis in Γ with an *induction identifier* α, and each atom in

A with a set M of identifiers indicating which hypotheses can be applied to the atom. Further details are explained in Section 4.

2. **Challenge:** In which order should the rules be applied?
   Our solution: We here adopt the following simple strategy, and evaluate it by experiments in Section 5.
   - Repeatedly apply the rule Apply$\bot$ if possible, until no new knowledge is obtained. (Even if the rule does not apply, applications of Induct and Unfold explained in the following items may make Apply$\bot$ applicable.)
   - If the knowledge cannot be updated by Apply$\bot$, select some atom from A in a breadth-first manner, and apply the rule Induct to the atom.
   - Apply the rule Unfold whenever Induct is applied.
   - Try to apply the rule Valid$\bot$ whenever the knowledge is updated.

3 Horn Constraint Solving Problems

This section formalizes Horn constraint solving problems and proves the correctness of our reduction from constraint solving to inductive theorem proving. We here restrict ourselves to constraint Horn clauses over the theory $\mathcal{T}_{\mathbb{Z}}$ of quantifier-free linear integer arithmetic for simplicity, although our induction-based Horn constraint solving method formalized in Section 4 supports constraints over any background theories supported by the underlying SMT solver. A *$\mathcal{T}_{\mathbb{Z}}$-formula* ϕ is a Boolean combination of atomic formulas $t_1 \leq t_2$, $t_1 < t_2$, $t_1 = t_2$, and $t_1 \neq t_2$. We write $\top$ and $\bot$ respectively for tautology and contradiction. A *$\mathcal{T}_{\mathbb{Z}}$-term* t is either a term variable x, an integer constant n, or $t_1 + t_2$.

3.1 Notation for HCSs

A *Horn Constraint Set (HCS)* $\mathcal{H}$ is a finite set $\{hc_1, \ldots, hc_m\}$ of Horn clauses. A *Horn clause* hc is defined to be $h \Leftarrow b$, consisting of a head h and a body b. A *head* h is either of the form $P(\widetilde{t})$ or $\bot$, and a *body* b is of the form $P_1(\widetilde{t_1}) \wedge \cdots \wedge P_m(\widetilde{t_m}) \wedge \phi$. Here, P is a meta-variable ranging over predicate variables. We write $ar(P)$ for the arity of P. We often abbreviate a Horn clause $h \Leftarrow \top$ as h. We write $pvs(hc)$ for the set of the predicate variables that occur in hc and define $pvs(\mathcal{H}) = \bigcup_{hc \in \mathcal{H}} pvs(hc)$. Similarly, we write $fvs(hc)$ for the set of the term variables in hc and define $fvs(\mathcal{H}) = \bigcup_{hc \in \mathcal{H}} fvs(hc)$. We assume that for any $hc_1, hc_2 \in \mathcal{H}$, $hc_1 \neq hc_2$ implies $fvs(hc_1) \cap fvs(hc_2) = \emptyset$. We write $\mathcal{H}\restriction_P$ for the set of Horn clauses in $\mathcal{H}$ of the form $P(\widetilde{t}) \Leftarrow b$. We define $\mathcal{H}(P) = \lambda\widetilde{x}.\exists\widetilde{y}.\bigvee_{i=1}^{m}(b_i \wedge \widetilde{x} = \widetilde{t_i})$ if $\mathcal{H}\restriction_P = \left\{P(\widetilde{t_i}) \Leftarrow b_i\right\}_{i \in \{1,\ldots,m\}}$ where $\{\widetilde{y}\} = fvs(\mathcal{H}\restriction_P)$ and $\{\widetilde{x}\} \cap \{\widetilde{y}\} = \emptyset$. By using $\mathcal{H}(P)$, an HCS $\mathcal{H}$ is logically interpreted as the formula $\bigwedge_{P \in pvs(\mathcal{H})} \forall \widetilde{x}_P.\, (\mathcal{H}(P)(\widetilde{x}_P) \Rightarrow P(\widetilde{x}_P))$, where $\widetilde{x}_P = x_1, \ldots, x_{ar(P)}$. A Horn clause with the head of the form $P(\widetilde{t})$ (resp. $\bot$) is called a *definite* clause (resp. a *goal* clause). We write $def(\mathcal{H})$ (resp. $goal(\mathcal{H})$) for the subset of $\mathcal{H}$ consisting of only the definite (resp. goal) clauses. Note that $\mathcal{H} = def(\mathcal{H}) \cup goal(\mathcal{H})$ and $def(\mathcal{H}) \cap goal(\mathcal{H}) = \emptyset$.

3.2 Predicate Interpretation

A *predicate interpretation* ρ for an HCS $\mathcal{H}$ is a map from each predicate variable $P \in pvs(\mathcal{H})$ to a subset of $\mathbb{Z}^{ar(P)}$. We write the domain of ρ as $\mathrm{dom}(\rho)$. We write $\rho_1 \subseteq \rho_2$ if $\rho_1(P) \subseteq \rho_2(P)$ for all $P \in pvs(\mathcal{H})$. We call an interpretation ρ a *solution of* $\mathcal{H}$ and write $\rho \models \mathcal{H}$ if $\rho \models hc$ holds for all $hc \in \mathcal{H}$. For example, $\rho'_{mult} = \{P \mapsto \{(x,y,r) \in \mathbb{Z}^3 \mid x \times y = r\}, Q \mapsto \{(x,y,a,r) \in \mathbb{Z}^4 \mid x \times y + a = r\}\}$ is a solution of the HCS $\mathcal{H}_{mult}$ in Section 1.

Definition 1 (Horn Constraint Solving Problems). *A* Horn constraint solving problem *is the problem of checking if a given HCS $\mathcal{H}$ has a solution.*

We now establish the reduction from Horn constraint solving to inductive theorem proving, which is the foundation of our induction-based Horn constraint solving method. The definite clauses $def(\mathcal{H})$ are considered to inductively define the *least predicate interpretation* for $\mathcal{H}$ as the least fixed-point $\mu F_{\mathcal{H}}$ of the following function on predicate interpretations.

$$F_{\mathcal{H}}(\rho) = \left\{ P \mapsto \left\{ (\widetilde{x}) \in \mathbb{Z}^{ar(P)} \;\middle|\; \rho \models \mathcal{H}(P)(\widetilde{x}) \right\} \;\middle|\; P \in \mathrm{dom}(\rho) \right\}$$

Because $F_{\mathcal{H}}$ is continuous [35], the least fixed-point $\mu F_{\mathcal{H}}$ of $F_{\mathcal{H}}$ exists. Furthermore, we can express it as $\mu F_{\mathcal{H}} = \bigcup_{i \in \mathbb{N}} F^i_{\mathcal{H}}(\{P \mapsto \emptyset \mid P \in pvs(\mathcal{H})\})$, where $F^i_{\mathcal{H}}$ means i-times application of $F_{\mathcal{H}}$. It immediately follows that the least predicate interpretation $\mu F_{\mathcal{H}}$ is a solution of $def(\mathcal{H})$ because any fixed-point of $F_{\mathcal{H}}$ is a solution of $def(\mathcal{H})$. Furthermore, $\mu F_{\mathcal{H}}$ is the least solution. Formally, we can prove the following proposition.

Proposition 1. *$\mu F_{\mathcal{H}} \models def(\mathcal{H})$ holds, and for all ρ such that $\rho \models def(\mathcal{H})$, $\mu F_{\mathcal{H}} \subseteq \rho$ holds.*

On the other hand, the goal clauses $goal(\mathcal{H})$ are considered as specifications of the least predicate interpretation $\mu F_{\mathcal{H}}$. As a corollary of Proposition 1, it follows that $\mathcal{H}$ has a solution if and only if $\mu F_{\mathcal{H}}$ satisfies the specifications $goal(\mathcal{H})$.

Corollary 1. *$\rho \models \mathcal{H}$ for some ρ if and only if $\mu F_{\mathcal{H}} \models goal(\mathcal{H})$*

In Section 4, we present an induction-based method for proving $\mu F_{\mathcal{H}} \models goal(\mathcal{H})$.

4 Induction-based Horn Constraint Solving Method

As explained in Section 2, our method is based on the reduction from Horn constraint solving into inductive theorem proving. The remaining task is to develop an automated method for proving the inductive conjectures obtained from Horn constraints. We thus formalize our inductive proof system tailored to Horn constraint solving and proves its soundness and relative completeness. To automate proof search in the system, we adopt the rule application strategy in Section 2.

We formalize a general and more elaborate version of the inductive proof system explained in Section 2. A judgment of the extended system is of the form

Perform induction on the derivation of the atom $P(\widetilde{t})$:

$$\frac{P^M_\circ(\widetilde{t}) \in A \qquad \Gamma' = \Gamma \cup \left\{(\alpha \triangleright P(\widetilde{t}), A, \phi, h)\right\} \qquad \mathcal{D}; \Gamma'; (A \setminus P^M_\circ(\widetilde{t})) \cup \left\{P^M_\alpha(\widetilde{t})\right\}; \phi \vdash h \qquad (\alpha : \text{fresh})}{\mathcal{D}; \Gamma; A; \phi \vdash h} \quad (\textsc{Induct})$$

Case-analyze the last rule used (where m rules are possible):

$$\frac{P^M_\alpha(\widetilde{t}) \in A \qquad \mathcal{D}(P)(\widetilde{t}) = \bigvee_{i=1}^m \exists \widetilde{x}_i.\, (\phi_i \wedge \bigwedge A_i) \qquad \mathcal{D}; \Gamma; A \cup A_{i\circ}^{M \cup \{\alpha\}}; \phi \wedge \phi_i \vdash h \qquad (\text{for each } i \in \{1, \ldots, m\})}{\mathcal{D}; \Gamma; A; \phi \vdash h} \quad (\textsc{Unfold})$$

Apply an induction hypothesis or a user-specified lemma in Γ:

$$\frac{(g, A', \phi', \bot) \in \Gamma \qquad \mathrm{dom}(\sigma) = \mathit{fvs}(A') \qquad \models \phi \Rightarrow [\![\sigma g \in A]\!] \qquad \models \phi \Rightarrow [\![\sigma A' \subseteq A]\!] \qquad \{\widetilde{x}\} = \mathit{fvs}(\phi') \setminus \mathrm{dom}(\sigma) \qquad \mathcal{D}; \Gamma; A; \phi \wedge \forall \widetilde{x}. \neg(\sigma\phi') \vdash h}{\mathcal{D}; \Gamma; A; \phi \vdash h} \quad (\textsc{Apply}\bot)$$

Apply an induction hypothesis or a user-specified lemma in Γ:

$$\frac{(g, A', \phi', P(\widetilde{t})) \in \Gamma \qquad \mathrm{dom}(\sigma) = \mathit{fvs}(A') \cup \mathit{fvs}(\widetilde{t}) \qquad \models \phi \Rightarrow [\![\sigma g \in A]\!] \qquad \models \phi \Rightarrow \exists \widetilde{x}.(\sigma\phi') \qquad \models \phi \Rightarrow [\![\sigma A' \subseteq A]\!] \qquad \{\widetilde{x}\} = \mathit{fvs}(\phi') \setminus \mathrm{dom}(\sigma) \qquad \mathcal{D}; \Gamma; A \cup \left\{P^\emptyset_\circ(\sigma\widetilde{t})\right\}; \phi \vdash h}{\mathcal{D}; \Gamma; A; \phi \vdash h} \quad (\textsc{Apply}P)$$

Apply a definite clause in $\mathcal{D}$:

$$\frac{(P(\widetilde{t}) \Leftarrow \phi' \wedge \bigwedge A') \in \mathcal{D} \qquad \mathrm{dom}(\sigma) = \mathit{fvs}(A') \cup \mathit{fvs}(\widetilde{t}) \qquad \models \phi \Rightarrow \exists \widetilde{x}.(\sigma\phi') \qquad \models \phi \Rightarrow [\![\sigma A' \subseteq A]\!] \qquad \{\widetilde{x}\} = \mathit{fvs}(\phi') \setminus \mathrm{dom}(\sigma) \qquad \mathcal{D}; \Gamma; A \cup \left\{P^\emptyset_\circ(\sigma\widetilde{t})\right\}; \phi \vdash h}{\mathcal{D}; \Gamma; A; \phi \vdash h} \quad (\textsc{Fold})$$

Check if the current knowledge entails the asserted proposition:

$$\frac{\mu F_\mathcal{D} \models \bigwedge A \wedge \phi \Rightarrow \bot}{\mathcal{D}; \Gamma; A; \phi \vdash \bot}\ (\textsc{Valid}\bot) \qquad \frac{\mu F_\mathcal{D} \models \bigwedge A \wedge \phi \Rightarrow [\![P(\widetilde{t}) \in A]\!]}{\mathcal{D}; \Gamma; A; \phi \vdash P(\widetilde{t})}\ (\textsc{Valid}P)$$

Auxiliary functions:

$$[\![P(\widetilde{t}) \in A]\!] \triangleq \bigvee_{P(\widetilde{t}') \in A} \widetilde{t} = \widetilde{t}' \qquad [\![A_1 \subseteq A_2]\!] \triangleq \bigwedge_{P(\widetilde{t}) \in A_1} [\![P(\widetilde{t}) \in A_2]\!]$$

$$[\![\bullet \in A]\!] \triangleq \top \qquad [\![\alpha \triangleright P(\widetilde{t}) \in A]\!] \triangleq \left[\!\left[P(\widetilde{t}) \in \left\{P^M(\widetilde{t}') \in A \mid \alpha \in M\right\}\right]\!\right]$$

Fig. 2. The inference rules for the judgment $\mathcal{D}; \Gamma; A; \phi \vdash h$.

$\mathcal{D}; \Gamma; A; \phi \vdash h$, where $\mathcal{D}$ is a set of definite clauses and ϕ represents a formula without atoms. We here assume that $\mathcal{D}(P)$ is defined similarly as $\mathcal{H}(P)$. The asserted proposition h on the right is now allowed to be an atom $P(\widetilde{t})$ instead of $\bot$. For deriving such judgments, we will introduce new rules Fold and Valid$P$ later in this section. Γ represents a set $\{(g_1, A_1, \phi_1, h_1), \ldots, (g_m, A_m, \phi_m, h_m)\}$ con-

sisting of user-specified lemmas and induction hypotheses, where g_i is either $\bullet$ or $\alpha \triangleright P(\widetilde{t})$. $(\bullet, A, \phi, h) \in \Gamma$ represents the user-specified lemma $\forall \widetilde{x}.\, (\bigwedge A \wedge \phi \Rightarrow h)$ where $\{\widetilde{x}\} = \mathit{fvs}(A, \phi, h)$, while $(\alpha \triangleright P(\widetilde{t}), A, \phi, h) \in \Gamma$ represents the induction hypothesis $\forall \widetilde{x}.\, \big(\big(P(\widetilde{t}) \prec P(\widetilde{t'})\big) \wedge \bigwedge A \wedge \phi \Rightarrow h\big)$ with $\{\widetilde{x}\} = \mathit{fvs}(P(\widetilde{t}), A, \phi, h)$ that has been introduced by induction on the derivation of the atom $P(\widetilde{t'})$. Here, α represents the *induction identifier* assigned to the application of induction that has introduced the hypothesis. Note that h on the right-hand side of $\Rightarrow$ is now allowed to be an atom of the form $Q(\widetilde{t})$. We will introduce a new rule ApplyP later in this section for using such lemmas and hypotheses to obtain new knowledge. A is also extended to be a set $\big\{P_1{}^{M_1}_{\alpha_1}(\widetilde{t}_1), \ldots, P_m{}^{M_m}_{\alpha_m}(\widetilde{t}_m)\big\}$ of annotated atoms. Each element $P^M_\alpha(\widetilde{t})$ has two annotations:

- an induction identifier α indicating that the induction with the identifier α is performed on the atom by the rule Induct. If the rule Induct has never been applied to the atom, α is set to be a special identifier denoted by $\circ$.
- a set of induction identifiers M indicating that if $\alpha' \in M$, the derivation D of the atom $P^M_\alpha(\widetilde{t})$ satisfies $D \prec D'$ for the derivation D' of the atom $P(\widetilde{t'})$ on which the induction with the identifier α' is performed. Thus, an induction hypothesis $(\alpha' \triangleright P(\widetilde{t'}), A', \phi', h') \in \Gamma$ can be applied to the atom $P^M_\alpha(\widetilde{t}) \in A$ only if $\alpha' \in M$ holds.

Note that we use these annotations only for guiding inductive proofs and $P^M_\alpha(\widetilde{t})$ is logically equivalent to $P(\widetilde{t})$. We often omit these annotations when they are clear from the context.

Given a Horn constraint solving problem $\mathcal{H}$, our method reduces the problem into an inductive theorem proving problem as follows. For each goal clause in $\mathit{goal}(\mathcal{H}) = \{\bigwedge A_i \wedge \phi_i \Rightarrow \bot\}_{i=1}^m$, we check the judgment $\mathit{def}(\mathcal{H}); \emptyset; A_i{}^{\emptyset}_{\circ}; \phi_i \vdash \bot$ is derivable by the inductive proof system. Here, each atom in A_i is initially annotated with $\emptyset$ and $\circ$.

The inference rules for the judgment $\mathcal{D}; \Gamma; A; \phi \vdash h$ are defined in Figure 2. The rule Induct selects an atom $P^M_\circ(\widetilde{t}) \in A$ and performs induction on the derivation of the atom. This rule generates a fresh induction identifier $\alpha \neq \circ$, adds a new induction hypothesis $(\alpha \triangleright P(\widetilde{t}), A, \phi, h)$ to Γ, and replaces the atom $P^M_\circ(\widetilde{t})$ with the annotated one $P^M_\alpha(\widetilde{t})$ for remembering that the induction with the identifier α is performed on it. The rule Unfold selects an atom $P^M_\alpha(\widetilde{t}) \in A$ and performs a case analysis on the last rule $P(\widetilde{t}) \Leftarrow \phi_i \wedge \bigwedge A_i$ used to derive the atom. As the result, the goal is broken into m-subgoals if there are m rules possibly used to derive the atom. The rule adds $A_i{}^{M \cup \{\alpha\}}_{\circ}$ and ϕ_i respectively to A and ϕ in the i-th subgoal, where A^M_α represents $\big\{P^M_\alpha(\widetilde{t}) \;\big|\; P(\widetilde{t}) \in A\big\}$. Note here that each atom in A_i is annotated with $M \cup \{\alpha\}$ because the derivation of the atom A_i is a strict sub-derivation of that of the atom $P^M_\alpha(\widetilde{t})$ on which the induction with the identifier α has been performed. If $\alpha = \circ$, it is the case that the rule Induct has never been applied to the atom $P^M_\alpha(\widetilde{t})$ yet. The rules Apply$\bot$ and ApplyP select $(g, A', \phi', h) \in \Gamma$, which represents a user-specified lemma if $g = \bullet$ and an induction hypothesis otherwise, and try to add new knowledge respectively to the ϕ- and the A-part of the current knowledge: the

rules try to find an instantiation σ for the free term variables in (g, A', ϕ', h), which are considered to be universally quantified, and then use $\sigma(g, A', \phi', h)$ to obtain new knowledge. Contrary to the rule UNFOLD, the rule FOLD tries to use a definite clause $P(\widetilde{t}) \Leftarrow \phi' \wedge \bigwedge A' \in \mathcal{D}$ from the body to the head direction: FOLD tries to find σ such that $\sigma(\phi' \wedge \bigwedge A')$ is implied by the current knowledge, and updates it with $P(\sigma\widetilde{t})$. This rule is useful when we check the correctness of user specified lemmas. The rule VALID$\bot$ checks if the current knowledge $\bigwedge A \wedge \phi$ implies a contradiction, while the rule VALIDP checks if the asserted proposition $P(\widetilde{t})$ on the right-hand side of the judgment is implied by the current knowledge. Here, we can use either an SMT solver or a (PDR-based) Horn constraint solver. The former is much faster because the validity checking problem $\mu F_{\mathcal{D}} \models \bigwedge A \wedge \phi \Rightarrow \psi$ is approximated to $\models \bigwedge \phi \Rightarrow \psi$. By contrast, the latter is much more precise because we reduce $\mu F_{\mathcal{D}} \models \bigwedge A \wedge \phi \Rightarrow \psi$ to Horn constraint solving of $\mathcal{D} \cup \{\bot \Leftarrow \bigwedge A \wedge \phi \wedge \neg\psi\}$. The soundness of the inductive proof system is shown as follows.

Lemma 1 (Soundness). *If $\mathcal{D}; \Gamma; A; \phi \vdash h$ is derivable, then there is k such that $\mu F_{\mathcal{D}} \models [\![\Gamma, A]\!]^k \wedge \bigwedge A \wedge \phi \Rightarrow h$ holds. Here, $[\![\Gamma, A]\!]^k$ represents the conjunction of user-specified lemmas and induction hypotheses in Γ instantiated for the atoms occurring in the k-times unfolding of A.*

The correctness of our Horn constraint solving method follows immediately from Lemma 1 and Corollary 1 as follows.

Theorem 1. *Suppose that $\mathcal{H}$ is an HCS with $goal(\mathcal{H}) = \{\bigwedge A_i \wedge \phi_i \Rightarrow \bot\}_{i=1}^m$ and $def(\mathcal{H}); \emptyset; A_i; \phi_i \vdash \bot$ for all $i = 1, \ldots, m$. It then follows $\rho \models \mathcal{H}$ for some ρ.*

Proof. By Lemma 1 and the fact that $\mu F_{def(\mathcal{H})} \models \bigwedge A_i \Rightarrow [\![\emptyset, A_i]\!]^k$, we get $\mu F_{def(\mathcal{H})} \models \bigwedge A_i \wedge \phi_i \Rightarrow \bot$. We therefore have $\mu F_{def(\mathcal{H})} \models goal(\mathcal{H})$. It then follows that $\rho \models \mathcal{H}$ for some ρ by Corollary 1.

We can also prove the following relative completeness of our system with respect to ordinary Horn constraint solving schemes that find solutions explicitly.

Lemma 2 (Relative Completeness). *Suppose that ρ is a solution of a given HCS $\mathcal{H}$ with $goal(\mathcal{H}) = \{\bot \Leftarrow \bigwedge A \wedge \phi\}$. Let $\Gamma = \{\phi_P \mid P \in \mathrm{dom}(\rho)\}$ where $\phi_P = (\forall\widetilde{x}.\ P(\widetilde{x}) \Rightarrow \rho(P)(\widetilde{x}))$. Then, $def(\mathcal{H}); \Gamma; A; \phi \vdash \bot$ and $def(\mathcal{H}); \Gamma \setminus \{\phi_P\}; P(\widetilde{x}); \neg\rho(P)(\widetilde{x}) \vdash \bot$ hold for all $P \in \mathrm{dom}(\rho)$.*

Note that our method can exploit over-approximations of the predicates computed by an existing Horn constraint solver as lemmas for checking the validity of the goal clauses, even if the existing solver failed to find a complete solution.

5 Implementation and Preliminary Experiments

We have implemented a Horn constraint solver based on the proposed method and integrated it, as a backend solver, with a refinement type-based verification tool RCaml [54, 55, 57] for the OCaml functional language. Our solver generates

a proof tree like the one shown in Figure 1 as a certificate if the given constraint set is judged to have a solution, and a counterexample if the constraint set is judged unsolvable. We adopted Z3 [22] and its PDR engine [32] respectively as the underlying SMT and Horn constraint solvers of the inductive proof system. In addition, our solver is extended to generate conjectures on the determinacy of the predicates, which are then checked and used as lemmas. This extension is particularly useful for verification of deterministic functions. The details of the implementation are explained in the full version [58]. The web interface to the verification tool as well as all the benchmark programs used in the experiments reported below are available from `http://www.cs.tsukuba.ac.jp/~uhiro/`.

We have tested our tool on two benchmark sets. The first set is obtained from the test suite for automated induction provided by the authors of the IsaPlanner system [23]. The benchmark set consists of 85 (mostly) relational verification problems of pure mathematical functions on algebraic data types (ADTs) such as natural numbers, lists, and binary trees. Most of the problems cannot be verified by using the previous Horn constraint solvers [27, 29, 33, 41, 48, 52, 55, 57] because they support neither relational verification nor ADTs. The benchmark set has also been used to evaluate the automated inductive theorem provers [18, 40, 46, 49]. The experiment results on this benchmark set are reported in Section 5.1.

To demonstrate advantages of our novel combination of Horn constraint solving with inductive theorem proving, we have prepared the second benchmark set consisting of 30 assertion safety verification problems of (mostly relational) specifications of OCaml programs that use various advanced language features such as partial (i.e., possibly non-terminating) functions, higher-order functions, exceptions, non-determinism, ADTs, and non-inductively defined data types (e.g., real numbers). The benchmark set also includes integer functions with complex recursion and a verification problem concerning the equivalence of programs written in different language paradigms. All the verification problems except 4 are relational ones where safe inductive invariants are not expressible in QF_LIA, and therefore not solvable by the previous Horn constraint solvers. These verification problems are naturally and automatically axiomatized by our method using predicates defined by Horn constraints as the least interpretation. By contrast, these problems cannot be straightforwardly axiomatized by the previous automated inductive theorem provers based on logics of pure total functions on ADTs. The experiment results on this benchmark set are reported in Section 5.2.

5.1 Experiments on IsaPlanner benchmark set

We manually translated the IsaPlanner benchmarks into assertion safety verification problems of OCaml programs, where we encoded natural numbers using integer primitives, and defined lists and binary trees as ADTs in OCaml. RCaml reduced the verification problems into Horn constraint solving problems using the constraint generation method proposed in [55]. Our solver then automatically solved 68 out of 85 verification problems without using lemmas, and 73 problems with the extension for conjecturing the determinacy of predicates enabled. We have manually analyzed the experiment results and found that 9 out of 12 failed

problems require lemmas. The other 3 problems caused timeout of Z3. It was because the rule application strategy implemented in our solver caused useless detours in proofs and put heavier burden on Z3 than necessary.

The experiment results on the IsaPlanner benchmark set show that our Horn-clause-based axiomatization of total recursive functions does not have significant negative impact on the automation of induction; According to the comparison in [49] of state-of-the-art automated inductive theorem provers, which are based on logics of pure total functions over ADTs, IsaPlanner [23] proved 47 out of the 85 IsaPlanner benchmarks, Dafny [40] proved 45, ACL2s [15] proved 74, and Zeno [49] proved 82. The HipSpec [18] inductive prover and the SMT solver CVC4 extended with induction [46] are reported to have proved 80. In contrast to our Horn-clause-based method, these inductive theorem provers can be, and in fact are directly applied to prove the conjectures in the benchmark set, because the benchmark set contains only pure total functions over ADTs.

It is also worth noting that, all the inductive provers that achieved better results than ours support automatic lemma discovery (beyond the determinacy), in a stark contrast to our solver. For example, the above result (80 out of 85) of CVC4 is obtained when they enable an automatic lemma discovery technique proposed in [46] and use a different encoding (called **dti** in [46]) of natural numbers than ours. When they disable the technique and use a similar encoding to ours (called **dtt** in [46]), CVC4 is reported to have proved 64. Thus, we believe that extending our method with automatic lemma discovery, which has been comprehensively studied by the automated induction community [15, 18, 34, 37, 46, 49], further makes induction-based Horn constraint solving powerful.

5.2 Experiments on benchmark set that uses advanced features

Table 1 summarizes the experiment results on the benchmark set. The column "specification" shows the verified specification and the column "kind" shows its kind, where "equiv", "assoc", "comm", "dist", "mono", "idem", "nonint", and "nonrel" respectively represent the equivalence, associativity, commutativity, distributivity, monotonicity, idempotency, non-interference, and non-relational. The column "features" shows the language features used in the verification problem, where each character has the following meaning: H: higher-order functions, E: exceptions, P: partial (i.e., possibly non-terminating) functions, D: demonic non-determinism, R: real functions, I: integer functions with complex recursion, N: nonlinear functions, C: procedures written in different programming paradigms. The column "result" represents whether our tool succeeded ✓or failed ✗. The column "time" represents the elapsed time for verification in seconds.

Our tool successfully solved 28 out of 30 problems. Overall, the results show that our tool can solve relational verification problems that use various advanced language features, in a practical time with surprisingly few user-specified lemmas. We also want to emphasize that the problem ID5, which required a lemma, is a relational verification problem involving two function calls with significantly different control flows: one recurses on x and the other on y. Thus, the result demonstrates an advantage of our induction-based method that it can exploit

Table 1. Experiment results on programs that use various language features

ID	specification	kind	features	result	time (sec.)
1	$\texttt{mult}\ x\ y + a = \texttt{mult\_acc}\ x\ y\ a$	equiv	P	✓	0.378
2	$\texttt{mult}\ x\ y = \texttt{mult\_acc}\ x\ y\ 0$	equiv	P	✓†	0.803
3	$\texttt{mult}\ (1+x)\ y = y + \texttt{mult}\ x\ y$	equiv	P	✓	0.403
4	$y \geq 0 \Rightarrow \texttt{mult}\ x\ (1+y) = x + \texttt{mult}\ x\ y$	equiv	P	✓	0.426
5	$\texttt{mult}\ x\ y = \texttt{mult}\ y\ x$	comm	P	✓‡	0.389
6	$\texttt{mult}\ (x+y)\ z = \texttt{mult}\ x\ z + \texttt{mult}\ y\ z$	dist	P	✓	1.964
7	$\texttt{mult}\ x\ (y+z) = \texttt{mult}\ x\ y + \texttt{mult}\ x\ z$	dist	P	✓	4.360
8	$\texttt{mult}\ (\texttt{mult}\ x\ y)\ z = \texttt{mult}\ x\ (\texttt{mult}\ y\ z)$	assoc	P	✗	n/a
9	$0 \leq x_1 \leq x_2 \wedge 0 \leq y_1 \leq y_2 \Rightarrow \texttt{mult}\ x_1\ y_1 \leq \texttt{mult}\ x_2\ y_2$	mono	P	✓	0.416
10	$\texttt{sum}\ x + a = \texttt{sum\_acc}\ x\ a$	equiv		✓	0.576
11	$\texttt{sum}\ x = x + \texttt{sum}\ (x-1)$	equiv		✓	0.452
12	$x \leq y \Rightarrow \texttt{sum}\ x \leq \texttt{sum}\ y$	mono		✓	0.593
13	$x \geq 0 \Rightarrow \texttt{sum}\ x = \texttt{sum\_down}\ 0\ x$	equiv	P	✓	0.444
14	$x < 0 \Rightarrow \texttt{sum}\ x = \texttt{sum\_up}\ x\ 0$	equiv	P	✓	0.530
15	$\texttt{sum\_down}\ x\ y = \texttt{sum\_up}\ x\ y$	equiv	P	✗	n/a
16	$\texttt{sum}\ x = \texttt{apply sum}\ x$	equiv	H	✓	0.430
17	$\texttt{mult}\ x\ y = \texttt{apply2 mult}\ x\ y$	equiv	H, P	✓	0.416
18	$\texttt{repeat}\ x\ (\texttt{add}\ x)\ a\ y\ = a + \texttt{mult}\ x\ y$	equiv	H, P	✓	0.455
19	$x \leq 101 \Rightarrow \texttt{mc91}\ x = 91$	nonrel	I	✓	0.233
20	$x \geq 0 \wedge y \geq 0 \Rightarrow \texttt{ack}\ x\ y > y$	nonrel	I	✓	0.316
21	$x \geq 0 \Rightarrow 2 \times \texttt{sum}\ x = x \times (x+1)$	nonrel	N	✓	0.275
22	$\texttt{dyn\_sys}\ 0.\ \not\rightarrow^* \texttt{assert false}$	nonrel	R,N	✓	0.189
23	$\texttt{flip\_mod}\ y\ x = \texttt{flip\_mod}\ y\ (\texttt{flip\_mod}\ y\ x)$	idem	P	✓	13.290
24	$\texttt{noninter}\ h_1\ l_1\ l_2\ l_3 = \texttt{noninter}\ h_2\ l_1\ l_2\ l_3$	nonint	P	✓	1.203
25	**try** $\texttt{find\_opt}\ p\ l = \texttt{Some}\ (\texttt{find}\ p\ l)$ **with** $\texttt{Not\_Found} \rightarrow \texttt{find\_opt}\ p\ l = \texttt{None}$	equiv	H, E	✓	1.065
26	**try** $\texttt{mem}\ (\texttt{find}\ ((=)\ x)\ l)\ l$ **with** $\texttt{Not\_Found} \rightarrow \neg(\texttt{mem}\ x\ l)$	equiv	H, E	✓	1.056
27	$\texttt{sum\_list}\ l = \texttt{fold\_left}\ (+)\ 0\ l$	equiv	H	✓	6.148
28	$\texttt{sum\_list}\ l = \texttt{fold\_right}\ (+)\ l\ 0$	equiv	H	✓	0.508
29	$\texttt{sum\_fun randpos}\ n > 0$	equiv	H,D	✓	0.319
30	$\texttt{mult}\ x\ y = \texttt{mult\_Ccode}(x, y)$	equiv	P, C	✓	0.303

† A lemma $P_{\texttt{mult\_acc}}(x, y, a, r) \Rightarrow P_{\texttt{mult\_acc}}(x, y, a-x, r-x)$ is used
‡ A lemma $P_{\texttt{mult}}(x, y, r) \Rightarrow P_{\texttt{mult}}(x-1, y, r-y)$ is used
Used a machine with Intel(R) Xeon(R) CPU (2.50 GHz, 16 GB of memory).

lemmas to fill the gap between function calls with different control flows. Another interesting result we obtain is that the distributivity ID7 of `mult` is solved thanks to our combination of inductive theorem proving and PDR-based Horn constraint solving, and just using either of them failed.

Our tool, however, failed to verify the associativity ID8 of `mult` and the equivalence ID15 of `sum_down` and `sum_up`. ID8 requires two lemmas $P_{\texttt{mult}}(x, y, r) \Rightarrow P_{\texttt{mult}}(y, x, r)$, which represents the commutativity of `mult`, and $P_{\texttt{mult}}(x{+}y, z, r) \Rightarrow \exists s_1, s_2.(P_{\texttt{mult}}(x, z, s_1) \wedge P_{\texttt{mult}}(y, z, s_2) \wedge r = s_1 + s_2)$. The latter lemma, however, is not of the form currently supported by our proof system. In ID15, the functions `sum_down` and `sum_up` use different recursion parameters (resp. y and x), and requires $P_{\texttt{sum\_down}}(x, y, s) \wedge a < x \Rightarrow \exists s_1, s_2.(P_{\texttt{sum\_down}}(a, y, s_1) \wedge P_{\texttt{sum\_down}}(a, x -$

$1, s_2) \wedge s = s_1 - s_2)$ and $P_{\texttt{sum\_up}}(x, y, s) \wedge a < x \Rightarrow \exists s_1, s_2.(P_{\texttt{sum\_down}}(a, y, s_1) \wedge P_{\texttt{sum\_down}}(a, x-1, s_2) \wedge s = s_1 - s_2)$. These lemmas are provable by induction on the derivation of $P_{\texttt{sum\_down}}(x, y, s)$ and $P_{\texttt{sum\_up}}(x, y, s)$, respectively. However, as in the case of ID8, our system does not support the form of the lemmas. Future work thus includes an extension to more general form of lemmas and judgments.

6 Related Work

Automated inductive theorem proving techniques and tools have long been studied, for example and to name a few: the Boyer-Moore theorem provers [37] ACL2s [15], rewriting induction provers [45] SPIKE [9], proof planners CLAM [14, 34] and IsaPlanner [23], and SMT-based induction provers Leon [50], Dafny [40], Zeno [49], HipSpec [18], and CVC4 extended with induction [46]. These automated provers are mostly based on logics of pure total functions over inductive data types. Consequently, users of these provers are required to axiomatize advanced language features and specifications using pure total functions as necessary. The axiomatization process, however, is non-trivial, error-prone, and possibly causes a negative effect on the automation of induction. For example, if a partial function (e.g., $f(x) = f(x) + 1$) is input, Zeno goes into an infinite loop and CVC4 is unsound (unless control literals proposed in [50] are used in the axiomatization). We have also confirmed that CVC4 failed to verify complex integer functions like the McCarthy 91 and the Ackermann functions (resp. ID19 and ID20 in Table 1). By contrast, our method supports advanced language features and specifications via Horn-clause encoding of their semantics based on program logics. Compared to cyclic proofs [11] and widely-supported structural induction on derivation trees, our proof system uses induction explicitly by maintaining a set of induction hypotheses and annotating atoms with induction identifiers so that we can apply the hypotheses soundly. This enables our system to introduce multiple induction hypotheses within a single proof path from dynamically generated formulas. Another advantage is the support of user-supplied lemmas, which are useful in relational verification involving function calls with different control flows (e.g., ID5). To address entailment checking problems in separation logic, researchers have recently proposed induction-based methods [12, 13, 16, 39, 51] to go beyond the popular unfold-and-match paradigm (see e.g. [44]). It seems fruitful to incorporate their techniques into our approach to Horn constraint solving to enable verification of heap-manipulating higher-order functional programs.

To aid verification of relational specifications of functional programs, Giesl [25] proposed context-moving transformations and Asada et al. [2] proposed a kind of tupling transformation. SymDiff [38] is a transformation-based tool built on top of Boogie [3] for equivalence verification of imperative programs. Self-composition [5] is a program transformation technique to reduce k-safety [19, 53] verification into ordinary safety verification, and has been applied to non-interference [4, 5, 53, 56] and regression verification [24] of imperative programs. These transformations are useful for some patterns of relational verification problems, which are, however, less flexible in some cases than our approach based on

a more general principle of induction. For example, Asada et al.'s transformation enables verification of the functional equivalence of recursive functions with the same recursion pattern (e.g., ID1), but does not help verification of the commutativity of `mult` (ID5). Because most of the transformations are designed for a particular target language, they cannot be applied to aid relational verification across programs written in different paradigms (e.g., ID30). Concurrently to our work, De Angelis et al. [1] recently proposed a predicate pairing transformation in the setting of Horn constraints for relational verification of imperative programs. We tested our tool with some of their benchmark constraint solving problems. There were some problems our tool successfully solved but their tool VeriMAP failed, and vice versa: only our tool solved "barthe2" in MON category (if its goal clause is generalized) and "product" in INJ category. We also confirmed that VeriMAP failed to solve our benchmark ID5 involving function calls with different control flows. On the contrary, VeriMAP solved ID15.

There have also been proposed program logics that allow precise relational verification [6, 7, 17, 26]. In particular, the relational refinement type system proposed in [6] can be applied to differential privacy and other relational security verification problems of higher-order functional programs. This approach, however, is not automated.

7 Conclusion and Future Work

We have proposed a novel Horn constraint solving method based on an inductive proof system and a PDR and SMT-based technique to automate proof search in the system. We have shown that our method can solve Horn clause constraints obtained from relational verification problems that were not possible with the previous methods based on Craig interpolation, abstract interpretation, and PDR. Furthermore, our novel combination of Horn clause constraints with inductive theorem proving enabled our method to automatically axiomatize and verify relational specifications of programs that use various advanced language features.

As a future work, we are planning to extend our inductive proof system to support more general form of lemmas and judgments. We are also planning to extend our proof search method to support automatic lemma discovery as in the state-of-the-art inductive theorem provers [15, 18, 46, 49]. To aid users to better understand verification results of our method, it is important to generate a symbolic representation of a solution of the original Horn constraint set from the found inductive proof. It is however often the case that a solution of Horn constraint sets that require relational analysis (e.g., $\mathcal{H}_{mult}$) is not expressible by a formula of the underlying logic. It therefore seems fruitful to generate a symbolic representation of mutual summaries in the sense of [31] across multiple predicates (e.g., P, Q of $\mathcal{H}_{mult}$).

Acknowledgments We would like to thank Tachio Terauchi for useful discussions, and anonymous referees for their constructive comments. This work was partially supported by Kakenhi 16H05856 and 15H05706.

References

1. E. D. Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Relational verification through horn clause transformation. In *SAS '16*, pages 147–169. Springer, 2016.
2. K. Asada, R. Sato, and N. Kobayashi. Verifying relational properties of functional programs by first-order refinement. In *PEPM '15*, pages 61–72. ACM, 2015.
3. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO '05*, pages 364–387. Springer, 2006.
4. G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In *FM '11*, pages 200–214. Springer, 2011.
5. G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *CSFW '04*, pages 100–114. IEEE, 2004.
6. G. Barthe, M. Gaboardi, E. Gallego Arias, J. Hsu, A. Roth, and P.-Y. Strub. Higher-order approximate relational refinement types for mechanism design and differential privacy. In *POPL '15*, pages 55–68. ACM, 2015.
7. G. Barthe, B. Köpf, F. Olmedo, and S. Zanella Béguelin. Probabilistic relational reasoning for differential privacy. In *POPL '12*, pages 97–110. ACM, 2012.
8. F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *Boogie '11*, pages 53–64, 2011.
9. A. Bouhoula, E. Kounalis, and M. Rusinowitch. SPIKE, an automatic theorem prover. In *LPAR '92*, volume 624 of *LNCS*, pages 460–462. Springer, 1992.
10. A. R. Bradley. SAT-based model checking without unrolling. In *VMCAI '11*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011.
11. J. Brotherston. Cyclic proofs for first-order logic with inductive definitions. In *TABLEAUX '05*, volume 3702 of *LNCS*, pages 78–92. Springer, 2005.
12. J. Brotherston, D. Distefano, and R. L. Petersen. Automated cyclic entailment proofs in separation logic. In *CADE-23*, pages 131–146. Springer, 2011.
13. J. Brotherston, C. Fuhs, J. A. P. Navarro, and N. Gorogiannis. A decision procedure for satisfiability in separation logic with inductive predicates. In *CSL-LICS '14*, pages 25:1–25:10. ACM, 2014.
14. A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In *CADE-10*, pages 647–648. Springer, 1990.
15. H. R. Chamarthi, P. Dillinger, P. Manolios, and D. Vroon. The ACL2 sedan theorem proving system. In *TACAS '11*, volume 6605 of *LNCS*, pages 291–295. Springer, 2011.
16. D.-H. Chu, J. Jaffar, and M.-T. Trinh. Automatic induction proofs of data-structures in imperative programs. In *PLDI '15*, pages 457–466. ACM, 2015.
17. Ş. Ciobâcă, D. Lucanu, V. Rusu, and G. Roşu. A language-independent proof system for mutual program equivalence. In *ICFEM '14*, pages 75–90. Springer, 2014.
18. K. Claessen, M. Johansson, D. Rosén, and N. Smallbone. Automating inductive proofs using theory exploration. In *CADE-24*, volume 7898 of *LNCS*, pages 392–406. Springer, 2013.

19. M. R. Clarkson and F. B. Schneider. Hyperproperties. In *CSF '08*, pages 51–65. IEEE, 2008.
20. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77*, pages 238–252. ACM, 1977.
21. W. Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Logic*, 22:269–285, 1957.
22. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS '08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
23. L. Dixon and J. D. Fleuriot. IsaPlanner: A prototype proof planner in Isabelle. In *CADE-19*, volume 2741 of *LNCS*, pages 279–283. Springer, 2003.
24. D. Felsing, S. Grebing, V. Klebanov, P. Rümmer, and M. Ulbrich. Automating regression verification. In *ASE '14*, pages 349–360. ACM, 2014.
25. J. Giesl. Context-moving transformations for function verification. In *LOPSTR '00*, volume 1817 of *LNCS*, pages 293–312. Springer, 2000.
26. B. Godlin and O. Strichman. Regression verification: proving the equivalence of similar programs. *Software Testing, Verification and Reliability*, 23(3):241–258, 2013.
27. S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI '12*, pages 405–416. ACM, 2012.
28. A. Gupta, C. Popeea, and A. Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In *POPL '11*, pages 331–344. ACM, 2011.
29. A. Gupta, C. Popeea, and A. Rybalchenko. Solving recursion-free horn clauses over LI+UIF. In *APLAS '11*, volume 7078 of *LNCS*, pages 188–203. Springer, 2011.
30. A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The SeaHorn verification framework. In *CAV '15*, pages 343–361. Springer, 2015.
31. C. Hawblitzel, M. Kawaguchi, S. K. Lahiri, and H. Rebêlo. Towards modularly comparing programs using automated theorem provers. In *CADE-24*, pages 282–299. Springer, 2013.
32. K. Hoder and N. Bjørner. Generalized property directed reachability. In *SAT '12*, pages 157–171. Springer, 2012.
33. K. Hoder, N. Bjørner, and L. de Moura. μZ: An efficient engine for fixed points with constraints. In *CAV '11*, volume 6806 of *LNCS*, pages 457–462. Springer, 2011.
34. A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1-2):79–111, 1996.
35. J. Jaffar and M. J. Maher. Constraint logic programming: a survey. *The Journal of Logic Programming*, 19:503 – 581, 1994.
36. T. Kahsai, P. Rümmer, H. Sanchez, and M. Schäf. JayHorn: A framework for verifying Java programs. In *CAV '16*, volume 9779, pages 352–358. Springer, 2016.
37. M. Kaufmann, J. S. Moore, and P. Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.
38. S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. SYMDIFF: A language-agnostic semantic diff tool for imperative programs. In *CAV '12*, pages 712–717. Springer, 2012.
39. Q. L. Le, J. Sun, and W.-N. Chin. Satisfiability modulo heap-based programs. In *CAV '16*, volume 9779, pages 382–404. Springer, 2016.
40. K. R. M. Leino. Automating induction with an SMT solver. In *VMCAI '12*, volume 7148 of *LNCS*, pages 315–331. Springer, 2012.
41. K. McMillan and A. Rybalchenko. Computing relational fixed points using interpolation. Technical Report MSR-TR-2013-6, Microsoft Research, 2013.

42. K. L. McMillan. An interpolating theorem prover. *Theoretical Computer Science*, 345(1):101–121, 2005.
43. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.
44. E. Pek, X. Qiu, and P. Madhusudan. Natural proofs for data structure manipulation in C using separation logic. In *PLDI '14*, pages 440–451. ACM, 2014.
45. U. S. Reddy. Term rewriting induction. In *CADE-10*, volume 449 of *LNCS*, pages 162–177. Springer, 1990.
46. A. Reynolds and V. Kuncak. Induction for SMT solvers. In *VMCAI '15*, volume 8931 of *LNCS*, pages 80–98. Springer, 2015.
47. P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI '08*, pages 159–169. ACM, 2008.
48. P. Rümmer, H. Hojjat, and V. Kuncak. Disjunctive interpolants for horn-clause verification. In *CAV '13*, volume 8044 of *LNCS*, pages 347–363. Springer, 2013.
49. W. Sonnex, S. Drossopoulou, and S. Eisenbach. Zeno: An automated prover for properties of recursive data structures. In *TACAS '12*, volume 7214 of *LNCS*, pages 407–421. Springer, 2012.
50. P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *SAS '11*, volume 6887 of *LNCS*, pages 298–315. Springer, 2011.
51. Q.-T. Ta, T. C. Le, S.-C. Khoo, and W.-N. Chin. Automated mutual explicit induction proof in separation logic. In *FM '16*, pages 659–676. Springer, 2016.
52. T. Terauchi. Dependent types from counterexamples. In *POPL '10*, pages 119–130. ACM, 2010.
53. T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *SAS '05*, volume 3672 of *LNCS*, pages 352–367. Springer, 2005.
54. H. Unno and N. Kobayashi. On-demand refinement of dependent types. In *FLOPS '08*, volume 4989 of *LNCS*, pages 81–96. Springer, 2008.
55. H. Unno and N. Kobayashi. Dependent type inference with interpolants. In *PPDP '09*, pages 277–288. ACM, 2009.
56. H. Unno, N. Kobayashi, and A. Yonezawa. Combining type-based analysis and model checking for finding counterexamples against non-interference. In *PLAS '06*, pages 17–26. ACM, 2006.
57. H. Unno and T. Terauchi. Inferring simple solutions to recursion-free horn clauses via sampling. In *TACAS '15*, volume 9035 of *LNCS*, pages 149–163. Springer, 2015.
58. H. Unno, S. Torii, and H. Sakamoto. Automating induction for solving horn clauses. Full version, available from `http://www.cs.tsukuba.ac.jp/~uhiro/`, 2017.
59. N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. L. Peyton Jones. Refinement types for Haskell. In *ICFP '14*, pages 269–282. ACM, 2014.
60. H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL '99*, pages 214–227. ACM, 1999.