

Automating Induction for Solving Horn Clauses

Hiroshi Unno Sho Torii

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

Abstract

Verification problems of programs written in various paradigms (such as imperative, logic, concurrent, functional, and object-oriented ones) 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 an SMT solver to discharge proof obligations arising in the proof search. The main advantage of the proposed method is that it 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. Furthermore, 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, non-deterministic, 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 [28], logic, concurrent [27], functional [45, 52, 53, 56], and object-oriented [35] 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 [25] 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 an SMT-based technique to automate proof search in the system.

Compared to previous Horn constraint solving methods [25, 26, 31, 41, 46, 50, 53, 54] based on Craig interpolation [18, 42], abstract interpretation [17], and PDR [10], the proposed method has two major advantages:

1. It can verify *relational specifications* where multiple function calls need to be analyzed simultaneously. As shown in Sections 3.3 and 5, the class of specifications includes practically

important ones such as functional equivalence, associativity, commutativity, distributivity, monotonicity, idempotency, and non-interference.

2. It can solve Horn clause constraints over whatever background theories supported by the underlying SMT solver. Example constraints in Section 3.3 are over the theories of nonlinear arithmetics and algebraic data structures, which have not been supported by available Horn constraint solvers to our knowledge.

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 functional program D_{mult} (in OCaml syntax).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 $\forall x, y, a \in \mathbb{N}. \text{main } x y a \not\rightarrow^* \text{assert false}$. By using existing Horn constraint generation methods for call-by-value functional programs [37, 53], the relational verification problem is reduced to the constraint solving problem of the following Horn clause constraint set \mathcal{H}_{mult} :

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

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 sec-

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 simple program as a running example.

ond 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 method [53] guarantees that the call-by-value 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} , $\perp \Leftarrow P(x, y, r_1) \wedge Q(x, y, a, r_2) \wedge (r_1 + a \neq r_2)$ that 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) arithmetic², 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 $P(x, y, r_1)$ and $Q(x, y, a, r_2)$. 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) of \mathcal{H}_{mult} , and uses induction on the derivation of $P(x, y, r_1)$ to prove the conjecture $\forall x, y, r_1, a, r_2. (P(x, y, r_1) \wedge Q(x, y, a, r_2) \wedge (r_1 + a \neq r_2) \Rightarrow \perp)$ represented by the goal clause (i.e., the clause whose head is *not* a predicate application) of \mathcal{H}_{mult} . Section 2 gives an overview of our method using this running example.

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 clause 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 Section 3.3 for more examples). Furthermore, we can automate the axiomatization process by using program logics such as Hoare logics for imperative and refinement type systems [45, 52, 53, 57] for functional programs. In fact, researchers have developed and made available tools such as SeaHorn [28] and JayHorn [35], respectively for translating C and Java programs into Horn clause constraints. In spite of the expressiveness, Horn clause constraints have a simpler logical semantics compared to other popular IVLs like Boogie [2] and Why3 [8]. This simplicity enabled us to directly apply inductive theorem proving and made the correctness proof and implementation easier.

In contrast to our induction method based on the logic of predicates defined by Horn clause constraints, most state-of-the-art automated inductive theorem provers such as ACL2s [13], Leon [49], Dafny [40], Zeno [48], HipSpec [15], and CVC4 [44] are based on

logics of pure total functions over inductively-defined data structures. Consequently, the axiomatization of advanced language features and specifications becomes a non-straightforward process, which often requires users' manual intervention and possibly has a negative effect on the automation of induction later. 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 formalizes Horn constraint solving problems and shows examples of the reduction from various program verification problems to Horn constraint solving problems. Section 4 formalizes our constraint solving method and proves its correctness. 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.

2. Overview of Induction-Based Horn Constraint Solving Method

In this section, we use the Horn constraint set \mathcal{H}_{mult} in Section 1 as a running example to give an overview of our induction-based Horn constraint solving method. Our method interprets the definite clauses (i.e., the clauses whose head is a predicate application) of a given Horn constraint set as derivation rules for predicate applications $P(\bar{t})$, which we call *atoms* henceforth. For example, the definite clauses $\mathcal{D}_{mult} \subseteq \mathcal{H}_{mult}$ are interpreted as the following derivation rules:

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

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

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. The above rules inductively define the least predicate interpretation $\{P \mapsto \{(x, y, r) \in \mathbb{Z}^3 \mid x \times y = r \wedge y \geq 0\}, Q \mapsto \{(x, y, a, r) \in \mathbb{Z}^4 \mid x \times y + a = r \wedge y \geq 0\}\}$ that satisfies the definite clauses \mathcal{D}_{mult} . It then follows that a given Horn constraint set has a solution if and only if all the goal clauses (i.e., the clauses whose head is *not* an atom) are valid under the interpretation (see Corollary 1 for the proof). Therefore, constraint solving of \mathcal{H}_{mult} boils down to the validity checking of the goal clause

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

under the least predicate interpretation for P and Q .

To check the validity of such a conjecture, our method uses 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 $\mathcal{D}; \Gamma; A; \phi \vdash \perp$, where \perp 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 predicate 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

²See <http://smt-lib.org/> for the definition of the theories.

$$\begin{array}{c}
\frac{P(\tilde{t}) \in A \quad \{\tilde{y}\} = \text{fvs}(A) \cup \text{fvs}(\phi) \quad \tilde{x} : \text{fresh} \quad \sigma = \{\tilde{y} \mapsto \tilde{x}\}}{\mathcal{D}; \Gamma \cup \{\forall \tilde{x}. ((P(\sigma\tilde{t}) \prec P(\tilde{t})) \wedge \bigwedge \sigma A \Rightarrow \neg(\sigma\phi))\}; A; \phi \vdash \perp} \\
\mathcal{D}; \Gamma; A; \phi \vdash \perp \quad (\text{INDUCT}) \\
\\
\frac{P(\tilde{t}) \in A \quad \sigma = \{\tilde{x} \mapsto \tilde{t}\} \quad \mathcal{D}; \Gamma; A \cup \sigma A'; \phi \wedge \sigma\phi' \vdash \perp \quad (\text{for each } (P(\tilde{x}) \leftarrow A' \wedge \phi') \in \mathcal{D})}{\mathcal{D}; \Gamma; A; \phi \vdash \perp} \quad (\text{UNFOLD}) \\
\\
\frac{\forall \tilde{x}. \left((P(\tilde{t}') \prec P(\tilde{t})) \wedge \bigwedge A' \Rightarrow \phi' \right) \in \Gamma \quad \text{dom}(\sigma) = \{\tilde{x}\} \quad \frac{P(\sigma\tilde{t}') \prec P(\tilde{t}) \quad \models \bigwedge A \wedge \phi \Rightarrow \bigwedge \sigma A'}{\mathcal{D}; \Gamma; A; \phi \wedge \sigma\phi' \vdash \perp}}{\mathcal{D}; \Gamma; A; \phi \vdash \perp} \quad (\text{APPLY} \perp) \\
\\
\frac{\models \phi \Rightarrow \perp}{\mathcal{D}; \Gamma; A; \phi \vdash \perp} \quad (\text{VALID} \perp)
\end{array}$$

Figure 1. A simplified version of the inference rules in Figure 3 for the judgment $\mathcal{D}; \Gamma; A; \phi \vdash \perp$.

scope within the clause, and are considered to be universally quantified (see Section 3 for a formal account). Intuitively, a judgment $\mathcal{D}; \Gamma; A; \phi \vdash \perp$ means that under the least predicate interpretation induced by \mathcal{D} , the formula $\bigwedge \Gamma \wedge \bigwedge A \wedge \phi \Rightarrow \perp$ is valid. For example, consider the following judgment J_{mult} :

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

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

The inference rules for the judgment $\mathcal{D}; \Gamma; A; \phi \vdash \perp$ are shown in Figure 3. 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 in Figure 1, 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** selects an atom $P(\tilde{t}) \in A$ and performs induction on the derivation of the atom by adding a new induction hypothesis $\forall \tilde{x}. ((P(\sigma\tilde{t}) \prec P(\tilde{t})) \wedge \bigwedge \sigma A \Rightarrow \neg(\sigma\phi))$ to Γ . Here, a map σ is used to generalize the free term variables \tilde{y} that occur in A or ϕ (denoted by $\text{fvs}(A) \cup \text{fvs}(\phi)$) into fresh variables \tilde{x} , and $P(\sigma\tilde{t}) \prec P(\tilde{t})$ requires that the derivation of $P(\sigma\tilde{t})$ is a strict sub-derivation of that of $P(\tilde{t})$. The second rule **UNFOLD** selects an atom $P(\tilde{t}) \in A$, performs a case analysis on the last rule used to derive the atom, which is represented by a definite clause in \mathcal{D} of the form $P(\tilde{x}) \leftarrow A' \wedge \phi'$, and updates the current knowledge $\bigwedge A \wedge \phi$ with $\bigwedge (A \cup \sigma A') \wedge \phi \wedge \sigma\phi'$ for $\sigma = \{\tilde{x} \mapsto \tilde{t}\}$. The third rule **APPLY** \perp selects an induction hypothesis in Γ , $\forall \tilde{x}. ((P(\tilde{t}') \prec P(\tilde{t})) \wedge \bigwedge A' \Rightarrow \phi')$, and tries to find an instantiation σ of the quantified variables \tilde{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 $P(\sigma\tilde{t}') \in \sigma A'$ to which the hypothesis is being applied is a strict sub-derivation of that of the atom

$$\begin{array}{c}
\frac{\overline{J_3} \text{ (VALID} \perp) \quad \overline{J_4} \text{ (VALID} \perp) \quad \overline{J_5} \text{ (VALID} \perp) \quad \overline{J_7} \text{ (VALID} \perp)}{\overline{J_6} \text{ (APPLY} \perp)} \\
\frac{\overline{J_3} \quad \overline{J_4} \quad \overline{J_5} \quad \overline{J_6} \quad \overline{J_7}}{J_1 \quad J_2 \quad J_6} \text{ (UNFOLD)} \\
\frac{J_0}{J_{\text{mult}}} \text{ (INDUCT)}
\end{array}$$

$$\begin{aligned}
J_0 &\triangleq \mathcal{D}_{\text{mult}}; \Gamma_{\text{mult}}; A_0; r_1 + a \neq r_2 \vdash \perp \\
J_1 &\triangleq \mathcal{D}_{\text{mult}}; \Gamma_{\text{mult}}; A_0; r_1 + a \neq r_2 \wedge y = 0 \wedge r_1 = 0 \vdash \perp \\
J_2 &\triangleq \mathcal{D}_{\text{mult}}; \Gamma_{\text{mult}}; A_P; r_1 + a \neq r_2 \wedge y \neq 0 \vdash \perp \\
J_3 &\triangleq \mathcal{D}_{\text{mult}}; \Gamma_{\text{mult}}; A_0; r_1 + a \neq r_2 \wedge y = 0 \wedge r_1 = 0 \wedge a = r_2 \vdash \perp \\
J_4 &\triangleq \mathcal{D}_{\text{mult}}; \Gamma_{\text{mult}}; A_Q; r_1 + a \neq r_2 \wedge y = 0 \wedge r_1 = 0 \wedge y \neq 0 \vdash \perp \\
J_5 &\triangleq \mathcal{D}_{\text{mult}}; \Gamma_{\text{mult}}; A_P; r_1 + a \neq r_2 \wedge y \neq 0 \wedge y = 0 \wedge a = r_2 \vdash \perp \\
J_6 &\triangleq \mathcal{D}_{\text{mult}}; \Gamma_{\text{mult}}; A_{PQ}; r_1 + a \neq r_2 \wedge y \neq 0 \vdash \perp \\
J_7 &\triangleq \mathcal{D}_{\text{mult}}; \Gamma_{\text{mult}}; A_{PQ}; r_1 + a \neq r_2 \wedge y \neq 0 \wedge r_1 + a = r_2 \vdash \perp \\
\Gamma_{\text{mult}} &\triangleq \{\forall x', y', r'_1, a', r'_2. ((P(x', y', r'_1) \prec P(x, y, r_1)) \wedge \\
&\quad P(x', y', r'_1) \wedge Q(x', y', a', r'_2) \Rightarrow r'_1 + a' = r'_2)\} \\
A_0 &\triangleq \{P(x, y, r_1), Q(x, y, a, r_2)\} \\
A_P &\triangleq A_0 \cup \{P(x, y - 1, r_1 - x)\} \\
A_Q &\triangleq A_0 \cup \{Q(x, y - 1, a + x, r_2)\} \\
A_{PQ} &\triangleq A_P \cup \{Q(x, y - 1, a + x, r_2)\}
\end{aligned}$$

Figure 2. The structure of an example derivation of J_{mult} .

$P(\tilde{t})$ on which the induction (that has introduced the hypothesis) has been performed.

If such a σ is found, the rule updates the current knowledge with $\bigwedge A \wedge \phi \wedge \sigma\phi'$. The fourth rule **VALID** \perp checks whether $\phi \Rightarrow \perp$ is valid, and if it is the case, closes the proof branch under consideration.

Figure 2 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 with an induction hypothesis Γ_{mult} 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 as the result, because $\mathcal{D}_{\text{mult}}$ 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 proved by the rule **VALID** \perp because $\models \phi_3 \Rightarrow \perp$ and $\models \phi_4 \Rightarrow \perp$ 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 proved by the rule **VALID** \perp because $\models \phi_5 \Rightarrow \perp$ holds. We then apply the induction hypothesis in Γ_{mult} ,

$$\begin{aligned}
&\forall x', y', r'_1, a', r'_2. ((P(x', y', r'_1) \prec P(x, y, r_1)) \wedge \\
&\quad P(x', y', r'_1) \wedge Q(x', y', a', r'_2) \Rightarrow r'_1 + a' = r'_2)
\end{aligned}$$

to the atom $P(x, y - 1, r_1 - x) \in A_{PQ}$ in J_6 using the rule **APPLY** \perp . Note that this can be done by using the quantifier

instantiation σ defined by

$$\{x' \mapsto x, y' \mapsto y - 1, r'_1 \mapsto r_1 - x, a' \mapsto a + x, r'_2 \mapsto r_2\},$$

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 $\bigwedge A_{PQ} \wedge r_1 + a \neq r_2 \wedge y \neq 0$. We thus obtain the subgoal J_7 , where the ϕ -part of the knowledge is updated to

$$\begin{aligned} & r_1 + a \neq r_2 \wedge y \neq 0 \wedge \sigma(r'_1 + a' = r'_2) \\ & \equiv r_1 + a \neq r_2 \wedge y \neq 0 \wedge (r_1 - x) + (a + x) = r_2 \\ & \equiv r_1 + a \neq r_2 \wedge y \neq 0 \wedge r_1 + a = r_2. \end{aligned}$$

Because this implies a contradiction, J_7 is finally proved by using the rule $\text{VALID}\perp$.

To automate proof search in the system, this paper proposes an SMT-based technique: we use an off-the-shelf SMT solver for checking whether the current knowledge implies a contradiction (in the rule $\text{VALID}\perp$) and whether there is an element of Γ that can be used to update the current knowledge, by finding a quantifier instantiation σ (in the rule $\text{APPLY}\perp$). The use of an SMT solver provides our method with efficient and powerful reasoning about data structures supported by SMT, including integers, real numbers, arrays, algebraic data types (ADTs), and uninterpreted functions. There, however, still remain two challenges to be addressed towards full automation:

1. **Challenge:** How to check (in the rule $\text{APPLY}\perp$) the strict sub-derivation relation $P(\tilde{t}') \prec P(\tilde{t})$ between the derivation of an atom $P(\tilde{t}')$ to which an induction hypothesis in Γ is being applied, and the derivation of the atom $P(\tilde{t})$ 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 $\text{APPLY}\perp$ 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: This paper adopts the following simple strategy, and evaluates it by experiments.

- Repeatedly apply the rule $\text{APPLY}\perp$ 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 $\text{APPLY}\perp$ applicable.)
- If the current knowledge cannot be updated by using the rule $\text{APPLY}\perp$, 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 $\text{VALID}\perp$ whenever the ϕ -part of the knowledge is updated.

3. Horn Constraint Solving Problems

This section formalizes Horn constraint solving problems and proves the correctness of our reduction from Horn constraint solving to inductive theorem proving in Corollary 1. Section 3.3 also shows example Horn constraint solving problems reduced from (relational) verification problems of programs that use various ad-

vanced language features, including higher-order functions and exceptions.

The syntax of Horn Clause Constraint Sets (HCCSs) over the theory $\mathcal{T}_{\mathbb{Z}}$ of quantifier-free linear integer arithmetic is defined by

$$\begin{aligned} (\text{HCCSs}) \mathcal{H} &::= \{hc_1, \dots, hc_m\} \\ (\text{Horn clauses}) hc &::= h \Leftarrow b \\ (\text{heads}) h &::= P(\tilde{t}) \mid \perp \\ (\text{bodies}) b &::= P_1(\tilde{t}_1) \wedge \dots \wedge P_m(\tilde{t}_m) \wedge \phi \\ (\mathcal{T}_{\mathbb{Z}}\text{-formulas}) \phi &::= t_1 \leq t_2 \mid \top \mid \perp \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \\ (\mathcal{T}_{\mathbb{Z}}\text{-terms}) t &::= x \mid n \mid t_1 + t_2 \end{aligned}$$

Here, the meta-variables P and x respectively represent predicate variables and term variables, and \tilde{t} represents a sequence of terms t_1, \dots, t_m . We write the arity of P as $\text{ar}(P)$. Note that, in the syntax of $\mathcal{T}_{\mathbb{Z}}$ -formulas, linear inequalities $t_1 \leq t_2$ can be used to encode $t_1 < t_2$, $t_1 = t_2$, and $t_1 \neq t_2$. For example, $t_1 < t_2$ is encoded as $t_1 + 1 \leq t_2$. The formula \top (resp. \perp) represents the tautology (resp. the contradiction). We here restrict ourselves to $\mathcal{T}_{\mathbb{Z}}$ for simplicity, although our induction-based Horn constraint solving method formalized in Section 4 supports constraints over whatever background theories supported by the underlying SMT solver, including the theories of nonlinear arithmetics, algebraic data structures, and uninterpreted function symbols as shown in Section 3.3.

3.1 Notation for HCCSs

A *Horn clause constraint set* \mathcal{H} is a finite set $\{hc_1, \dots, hc_m\}$ of Horn clauses. A *Horn clause* $h \Leftarrow b$ consists of a head h and a body b . We often abbreviate a Horn clause $h \Leftarrow \top$ as h . We write $\text{pvs}(hc)$ for the set of the predicate variables that occur in hc and define $\text{pvs}(\mathcal{H}) = \bigcup_{hc \in \mathcal{H}} \text{pvs}(hc)$. Similarly, we write $\text{fvs}(hc)$ for the set of the term variables in hc and define $\text{fvs}(\mathcal{H}) = \bigcup_{hc \in \mathcal{H}} \text{fvs}(hc)$. We assume that for any $hc_1, hc_2 \in \mathcal{H}$, $hc_1 \neq hc_2$ implies $\text{fvs}(hc_1) \cap \text{fvs}(hc_2) = \emptyset$. We write $\mathcal{H} \upharpoonright_P$ for the set of Horn clauses in \mathcal{H} of the form $P(\tilde{t}) \Leftarrow b$. We define $\mathcal{H}(P) = \lambda \tilde{x}. \exists \tilde{y}. \bigvee_{i=1}^m (b_i \wedge \tilde{x} = \tilde{t}_i)$ if $\mathcal{H} \upharpoonright_P = \{P(\tilde{t}_i) \Leftarrow b_i\}_{i \in \{1, \dots, m\}}$ where $\{\tilde{y}\} = \text{fvs}(\mathcal{H} \upharpoonright_P)$ and $\{\tilde{x}\} \cap \{\tilde{y}\} = \emptyset$. By using $\mathcal{H}(P)$, an HCCS \mathcal{H} is logically interpreted as the formula

$$\bigwedge_{P \in \text{pvs}(\mathcal{H})} \forall \tilde{x}_P. (\mathcal{H}(P)(\tilde{x}_P) \Rightarrow P(\tilde{x}_P)),$$

where $\tilde{x}_P = x_1, \dots, x_{\text{ar}(P)}$. A Horn clause with the head of the form $P(\tilde{t})$ (resp. \perp) is called a *definite clause* (resp. a *goal clause*). We write $\text{def}(\mathcal{H})$ (resp. $\text{goal}(\mathcal{H})$) for the subset of \mathcal{H} consisting of only the definite (resp. goal) clauses. Note that $\mathcal{H} = \text{def}(\mathcal{H}) \cup \text{goal}(\mathcal{H})$ and $\text{def}(\mathcal{H}) \cap \text{goal}(\mathcal{H}) = \emptyset$.

3.2 Predicate Interpretation

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

Definition 1 (Horn Constraint Solving Problems). A Horn constraint solving problem is the problem of checking whether a given HCCS \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 $\text{def}(\mathcal{H})$ 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.

$$F_{\mathcal{H}}(\rho) = \left\{ P \mapsto \left\{ (\tilde{x}) \in \mathbb{Z}^{ar(P)} \mid \rho \models \mathcal{H}(P)(\tilde{x}) \right\} \mid P \in \text{dom}(\rho) \right\}$$

Because $F_{\mathcal{H}}$ is continuous [33], the least fixed-point $\mu F_{\mathcal{H}}$ of $F_{\mathcal{H}}$ exists. Furthermore, we can express it as

$$\mu F_{\mathcal{H}} = \bigcup_{i \in \mathbb{N}} F_{\mathcal{H}}^i(\{P \mapsto \emptyset \mid P \in \text{pvs}(\mathcal{H})\}),$$

where $F_{\mathcal{H}}^i$ means i -times application of $F_{\mathcal{H}}$. It immediately follows that the least predicate interpretation $\mu F_{\mathcal{H}}$ is a solution of $\text{def}(\mathcal{H})$ because any fixed-point of $F_{\mathcal{H}}$ is a solution of $\text{def}(\mathcal{H})$. Furthermore, $\mu F_{\mathcal{H}}$ is the least solution. Formally, we can prove the following proposition.

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

On the other hand, the goal clauses $\text{goal}(\mathcal{H})$ 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 $\text{goal}(\mathcal{H})$.

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

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

3.3 Examples Reduced from Program Verification Problems

This section shows example Horn constraint solving problems reduced from (relational) verification problems of programs that use advanced language features such as algebraic data structures, higher-order functions, and exceptions. The reduction used in this section is mostly based on an existing Horn constraint generation method [53] for an ML-like (i.e., call-by-value, statically-typed, and higher-order) functional language. The method can be used to reduce a given assertion safety verification problem defined below into a Horn constraint solving problem.

Definition 2 (Assertion Safety Verification Problems). *An assertion safety verification problem of a given functional program, with a special function `main` of the ordinary ML type $\text{int} \rightarrow \dots \rightarrow \text{int} \rightarrow \text{unit}$, is the problem of deciding whether*

$$\forall n_1, \dots, n_m \in \mathbb{Z}. \text{main } n_1 \dots n_m \not\rightarrow^* \text{assert false},$$

where \rightarrow^* is the one-step evaluation relation. We call the program safe if this property holds, and unsafe otherwise.

The constraint generation method is based on refinement types [57], which are used internally to express value dependent inductive invariants and specifications of the program. The following theorem states the soundness of the reduction.

Theorem 1 (Soundness [53]). *Let \mathcal{H} be the HCCS generated from a program D . If there exists a solution ρ of \mathcal{H} , then D is safe.*

We now show example Horn constraints generated by the method. The partial recursive functions shown in Section 1 are automatically axiomatized using refinement types as follows.

Example 1. Recall the program D_{mult} in Section 1. The constraint generation method first prepares the following refinement type tem-

plates for the functions in D_{mult} .

$$\begin{aligned} \text{mult} &: (x : \text{int}) \rightarrow (y : \text{int}) \rightarrow \{r : \text{int} \mid P(x, y, r)\} \\ \text{mult_acc} &: (x : \text{int}) \rightarrow (y : \text{int}) \rightarrow (a : \text{int}) \rightarrow \\ &\quad \{r : \text{int} \mid Q(x, y, a, r)\} \end{aligned}$$

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 constraint generation method then type-checks the program against the type templates, and obtains the Horn constraint set $\mathcal{H}_{\text{mult}}$ in Section 1, which has a solution if and only if the program is typable under a refinement type system. The refinement type system guarantees that if a given program is typable, the evaluation of `main` never causes an assertion failure for any integer n . \square

Example 2. Consider the following program.

```
let rec sum n =
  if n < 0 then n + sum (n + 1)
  else if n = 0 then 0 else n + sum (n - 1)
let rec sum_acc n a =
  if n < 0 then sum_acc (n + 1) (a + n)
  else if n = 0 then a else sum_acc (n - 1) (a + n)
let main n a = assert(sum n + a = sum_acc n a)
```

In a similar manner to Example 1, we obtain the following Horn constraint set:

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

Here, the predicate variable P (resp. Q) represents an inductive invariant among the arguments and the return value of the function `sum` (resp. `sum_acc`). Here, suppose that the main function is replaced by

```
let main n =
  if n >= 0 then assert (2 * sum n = n * (n + 1))
```

We then obtain the following goal clause over the nonlinear integer arithmetic instead:

$$\perp \Leftarrow P(x, r) \wedge x \geq 0 \wedge 2 \times r \neq x \times (x + 1)$$

\square

The method can automatically axiomatize complex recursive functions on integers.

Example 3. Consider the following program with complex recursion.

```
let rec mc91 x =
  if x > 100 then x - 10 else mc91 (mc91 (x + 11))
let main x = if x <= 101 then assert(mc91 x = 91)
```

By using the refinement type template

$$\text{mc91} : (x : \text{int}) \rightarrow \{r : \text{int} \mid P(x, r)\}$$

the constraint generation method returns the following Horn constraint set:

$$\left\{ \begin{array}{l} P(x, x - 10) \Leftarrow x > 100, \\ P(x, s) \Leftarrow P(x + 11, r) \wedge P(r, s) \wedge x \leq 100, \\ \perp \Leftarrow P(x, r) \wedge x \leq 101 \wedge r \neq 91 \end{array} \right\}$$

Here, the predicate variable P represents an inductive invariant among the arguments and the return value of the function `mc91`. \square

Our method can also handle recursive functions on non-inductively defined data types such as real numbers.

Example 4. Consider the following program that models a dynamical system from [38].

```
let rec dyn_sys vc =
  let fa = 0.5418 *. vc *. vc in (* the force control *)
  let fr = 1000. -. fa in
  let ac = 0.0005 *. fr in
  let vc' = vc +. ac in
  assert (vc' < 49.61); (* the safety velocity *)
  dyn_sys vc'
let main () = dyn_sys 0. (* the initial velocity *)
```

By using the refinement type template

$\text{dyn_sys} : \{x : \text{real} \mid P(x)\} \rightarrow \text{unit}$

the constraint generation method returns the following Horn constraint set:

$$\left\{ \begin{array}{l} P(vc') \Leftarrow P(vc) \wedge fa = 0.5418 \times vc \times vc \wedge fr = 1000 - fa \wedge \\ \quad ac = 0.0005 \times fr \wedge vc' = vc + ac \wedge vc' < 49.61, \\ P(0), \\ \perp \Leftarrow P(vc) \wedge fa = 0.5418 \times vc \times vc \wedge fr = 1000 - fa \wedge \\ \quad ac = 0.0005 \times fr \wedge vc' = vc + ac \wedge vc' \geq 49.61 \end{array} \right\}$$

Here, the predicate variable P represents an inductive invariant on the argument of the function dyn_sys . \square

The constraint generation method can handle functional programs that manipulate user-defined algebraic data structures.

Example 5. Consider the following program that manipulates lists.

```
type list = Nil | Cons of int * list

let rec append l ys = match l with
| Nil -> ys
| Cons(x, xs) -> Cons(x, append xs ys)
let rec drop n l = match l with
| Nil -> Nil
| Cons(x, xs) ->
  if n = 0 then Cons(x, xs) else drop (n - 1) xs
let rec take n l = match l with
| Nil -> Nil
| Cons(x, xs) ->
  if n = 0 then Nil else Cons(x, take (n - 1) xs)
let main n xs =
  assert(append (take n xs) (drop n xs) = xs)
```

By using the refinement type templates

$\begin{array}{l} \text{append} : (x : \text{list}) \rightarrow (y : \text{list}) \rightarrow \{r : \text{list} \mid P(x, y, r)\} \\ \text{drop} : (x : \text{int}) \rightarrow (y : \text{list}) \rightarrow \{r : \text{list} \mid Q(x, y, r)\} \\ \text{take} : (x : \text{int}) \rightarrow (y : \text{list}) \rightarrow \{r : \text{list} \mid R(x, y, r)\} \end{array}$

the constraint generation method returns the following Horn constraint set over the theory of algebraic data structures:

$$\left\{ \begin{array}{l} P(\text{Nil}, l_2, l_2), \\ P(\text{Cons}(x, l), l_2, \text{Cons}(x, r)) \Leftarrow P(l, l_2, r), \\ Q(n, \text{Nil}, \text{Nil}), \\ Q(n, \text{Cons}(x, l'), \text{Cons}(x, l')) \Leftarrow n = 0, \\ Q(n, \text{Cons}(x, l'), r) \Leftarrow Q(n - 1, l', r) \wedge n \neq 0, \\ R(n, \text{Nil}, \text{Nil}), \\ R(n, \text{Cons}(x, l'), \text{Nil}) \Leftarrow n = 0, \\ R(n, \text{Cons}(x, l'), \text{Cons}(x, r)) \Leftarrow Q(n - 1, l', r) \wedge n \neq 0, \\ \perp \Leftarrow P(n, l, r_1) \wedge Q(n, l, r_2) \wedge R(r_1, r_2, r) \wedge r \neq l \end{array} \right\}$$

\square

The method can also axiomatize higher-order functions into Horn clause constraints automatically.

Example 6. Consider the following higher-order program.

```
type list = Nil | Cons of int * list

let rec sum_list l = match l with
| Nil -> 0
| Cons(x, xs) -> x + sum_list xs
let rec fold_left f s l = match l with
| Nil -> s
| Cons(x, xs) -> fold_left f (f s x) xs
let plus x y = x + y
let main l = assert(sum_list l = fold_left plus 0 l)
```

By using the refinement type templates

$\begin{array}{l} \text{sum_list} : (x : \text{list}) \rightarrow \{r : \text{int} \mid P(x, r)\} \\ \text{fold_left} : (f : (a : \text{int}) \rightarrow (b : \text{int}) \rightarrow \{c : \text{int} \mid Q(a, b, c)\}) \\ \quad \rightarrow (x : \text{int}) \rightarrow (y : \text{list}) \rightarrow \{z : \text{int} \mid R(x, y, z)\} \\ \text{plus} : (x : \text{int}) \rightarrow (y : \text{list}) \rightarrow \{r : \text{list} \mid S(x, y, r)\} \end{array}$

the constraint generation method returns the following Horn constraint set over the theories of linear integer arithmetic and algebraic data structures:

$$\left\{ \begin{array}{l} P(\text{Nil}, 0), \\ P(\text{Cons}(x, l'), x + r) \Leftarrow P(l', r), \\ R(s, \text{Nil}, s), \\ R(s, \text{Cons}(x, l'), r') \Leftarrow Q(s, x, r) \wedge R(r, l', r'), \\ S(x, y, x + y), \\ Q(x, y, z) \Leftarrow S(x, y, z), \\ \perp \Leftarrow P(l, r_1) \wedge R(0, l, r_2) \wedge r_1 \neq r_2 \end{array} \right\}$$

\square

The method can also axiomatize recursive functions that may raise exceptions into Horn clause constraints.

Example 7. Consider the following higher-order program that manipulates lists and possibly raises and catches exceptions.

```
exception Not_found
type int_option = None | Some of int

let rec find p l = match l with
| [] -> raise Not_found
| x::xs -> if p x then x else find p xs
let rec find_opt p l = match l with
| [] -> None
| x::xs -> if p x then Some x else find_opt p xs
let main p l = try find_opt p l = Some (find p l)
  with Not_found -> find_opt p l = None
```

Here, find and find_opt respectively use the exception Not_found and the option type int_option , defined respectively in the first and the second lines, for finding the first element of the list l satisfying the predicate $p : \text{int} \rightarrow \text{bool}$. The constraint generation method cannot directly handle this program because the underlying refinement type system does not support exceptions. Note, however, that we can mechanically transform the program into the following one by eliminating exceptions using a selective CPS transformation [47].

```
type exc = Not_found
type int_option = None | Some of int

let rec find p l ok ex = match l with
| [] -> ex Not_found
| x::xs -> if p x then ok x else find p xs ok ex
let rec find_opt p l = match l with
```

```

| [] -> None
| x::xs -> if p x then Some x else find_opt p xs
let main p l =
  find p l (fun x -> assert (find_opt p l = Some x))
  (fun Not_found -> assert (find_opt p l = None))

```

Note here that two function arguments `ok` and `ex` of the ordinary ML type `int → unit`, which respectively represent continuations for normal and exceptional cases, are added to the function `find`. The constraint generation method then prepares the following refinement type templates:

```

find : (x : int → bool) → (y : list) →
  ({z : int | Pok(x, y, z)} → unit) →
  ({w : exc | Pex(x, y, w)} → unit) → unit
find_opt : (x : int → bool) → (y : list) →
  {z : int_option | Q(x, y, z)}

```

Here, the predicate variable P_{ok} represents invariants among the first and the second arguments of `find` and the argument of the third argument `ok` of `find`. The predicate variable P_{ex} represents invariants among the first and the second arguments of `find` and the argument of the fourth argument `ex` of `find`. The predicate variable Q represents invariants among the arguments and the return value of `find_opt`. The constraint generation method then obtains the following Horn constraint set over the theories of linear integer arithmetic, algebraic data structures, and uninterpreted function symbols:

$$\left\{ \begin{array}{l} P_{ok}(p, x :: xs, r) \Leftarrow p x = \top, \\ P_{ok}(p, x :: xs, r) \Leftarrow P_{ok}(p, xs, r) \wedge p x = \perp, \\ P_{ex}(p, [], \text{Not_Found}), \\ P_{ex}(p, x :: xs, r) \Leftarrow P_{ex}(p, xs, r) \wedge p x = \perp, \\ Q(p, [], \text{None}), \\ Q(p, x :: xs, \text{Some } x) \Leftarrow p x = \top, \\ Q(p, x :: xs, r) \Leftarrow Q(p, xs, r) \wedge p x = \perp, \\ \perp \Leftarrow P_{ok}(p, l, r_1) \wedge Q(p, l, r_2) \wedge \text{Some } r_1 \neq r_2 \\ \perp \Leftarrow P_{ex}(p, l, r_1) \wedge r_1 \neq \text{Not_Found} \\ \perp \Leftarrow P_{ex}(p, l, \text{Not_Found}) \wedge Q(p, l, r_2) \wedge \text{None} \neq r_2 \end{array} \right\}$$

Here, p is an uninterpreted function symbol, which is essential for the success of Horn constraint solving here because we need to express the fact that the multiple occurrences of $p x$ in the body of different clauses return the same value if the same function is passed as p . \square

Our method also supports demonic non-determinism.

Example 8. Consider the following higher-order program that calls `rand_int` to generate random integers.

```

let rec randpos dummy =
  let n = rand_int () in
  if n > 0 then n else randpos dummy
let rec sum_fun f n =
  if n = 0 then f 0
  else f n + sum_fun f (n - 1)
let main n = assert (sum_fun randpos n > 0)

```

Note that the specification is satisfied because the function `randpos` never returns a non-positive integer. By using the refinement type templates

```

randpos : (x : int) → {y : list | P(x, y)}
sum_fun : (f : (a : int) → {b : int | Q(a, b)}) →
  (x : int) → {y : int | R(f, x, y)}

```

we obtain the following Horn constraint set:

$$\left\{ \begin{array}{l} P(x, y) \Leftarrow y > 0, \\ P(x, y) \Leftarrow P(x, y) \wedge y \leq 0, \\ Q(a, b) \Leftarrow Q(x, r) \wedge Q(a, b) \wedge x \neq 0, \\ Q(a, b) \Leftarrow P(a, b), \\ R(f, 0, y) \Leftarrow Q(0, y), \\ R(f, x, r_1 + r_2) \Leftarrow \\ \quad Q(0, y) \wedge Q(x, r_1) \wedge R(f, x - 1, r_2) \wedge x \neq 0, \\ \perp \Leftarrow Q(\text{randpos}, x, y) \wedge y \leq 0 \end{array} \right\}$$

\square

Our method based on Horn clause constraints is not limited to relational verification of functional programs. By combining the constraint generation tools for C [28] and Java [35], we can axiomatize relational verification problems across functional, imperative, object-oriented, and, of course, (constraint) logic programs into Horn clause constraints.

Example 9. Consider the following C program.

```

int mult(int x, int y) {
  int r=0; while(y != 0) r = r + x; y = y - 1;
  return r;
}

```

Using the Hoare logic, we obtain the following Horn constraint set:

$$\left\{ \begin{array}{l} I(x, y, r) \Leftarrow r = 0, \\ I(x, y - 1, r + x) \Leftarrow I(x, y, r) \wedge y \neq 0, \\ R(x, y, r) \Leftarrow I(x, y, r) \wedge y = 0 \end{array} \right\}$$

Here, the predicate variable I represents the loop invariant of the while loop, and R represents invariants among the arguments x, y and the return value r of the procedure `mult`. The goal clause $\perp \Leftarrow P(x, y, r_1) \wedge R(x, y, r_2) \wedge r_1 \neq r_2$, with the predicate P defined by $\mathcal{H}_{\text{mult}}$ in Section 1, represents the equivalence of C and OCaml implementations of `mult`. \square

It is also worth mentioning here that there have also been proposed techniques for reducing verification problems of multi-threaded programs [25, 27] and functional programs with the call-by-need evaluation strategy [56] into Horn constraint solving problems. Angelic non-determinism [29] and temporal specifications [7] can also be automatically axiomatized into Horn clause constraints extended with existentially quantified heads.

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 correctness of the reduction is established by Corollary 1 in Section 3. The remaining task is to develop an automated method for proving the inductive conjectures obtained from Horn clause constraints. To this end, Section 4.1 formalizes our inductive proof system tailored to Horn constraint solving and proves its correctness. Section 4.2 discusses how to automate proof search in the system using an SMT solver.

4.1 Inductive Proof System

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 $\mathcal{D}; \Gamma; A; \phi \vdash h$, 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 $P(\bar{t})$ instead of \perp . For deriving such judgments, we will introduce new rules FOLD and VALIDP later in this section. Γ represents a

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

$$\frac{P_{\circ}^M(\tilde{t}) \in A \quad \Gamma' = \Gamma \cup \{(\alpha \triangleright P(\tilde{t}), A, \phi, h)\} \quad \mathcal{D}; \Gamma'; (A \setminus P_{\circ}^M(\tilde{t})) \cup \{P_{\alpha}^M(\tilde{t})\}; \phi \vdash h \quad (\alpha : \text{fresh})}{\mathcal{D}; \Gamma; A; \phi \vdash h} \text{ (INDUCT)}$$

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

$$\frac{P_{\alpha}^M(\tilde{t}) \in A \quad \mathcal{D}(P)(\tilde{t}) = \bigvee_{i=1}^m \exists \tilde{x}_i. (\phi_i \wedge \bigwedge A_i) \quad \mathcal{D}; \Gamma; A \cup A_{i_{\circ}}^{M \cup \{\alpha\}}; \phi \wedge \phi_i \vdash h \quad (\text{for each } i \in \{1, \dots, m\})}{\mathcal{D}; \Gamma; A; \phi \vdash h} \text{ (UNFOLD)}$$

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

$$\frac{(g, A', \phi', \perp) \in \Gamma \quad \text{dom}(\sigma) = \text{fvs}(A') \quad \vdash \phi \Rightarrow \llbracket \sigma g \in A \rrbracket \quad \vdash \phi \Rightarrow \llbracket \sigma A' \subseteq A \rrbracket \quad \{\tilde{x}\} = \text{fvs}(\phi') \setminus \text{dom}(\sigma) \quad \mathcal{D}; \Gamma; A; \phi \wedge \forall \tilde{x}. \neg(\sigma \phi') \vdash h}{\mathcal{D}; \Gamma; A; \phi \vdash h} \text{ (APPLY}\perp\text{)}$$

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

$$\frac{(g, A', \phi', P(\tilde{t})) \in \Gamma \quad \text{dom}(\sigma) = \text{fvs}(A') \cup \text{fvs}(\tilde{t}) \quad \vdash \phi \Rightarrow \llbracket \sigma g \in A \rrbracket \quad \vdash \phi \Rightarrow \exists \tilde{x}. (\sigma \phi') \quad \vdash \phi \Rightarrow \llbracket \sigma A' \subseteq A \rrbracket \quad \{\tilde{x}\} = \text{fvs}(\phi') \setminus \text{dom}(\sigma) \quad \mathcal{D}; \Gamma; A \cup \{P_{\circ}^{\emptyset}(\sigma \tilde{t})\}; \phi \vdash h}{\mathcal{D}; \Gamma; A; \phi \vdash h} \text{ (APPLY}P\text{)}$$

Apply a definite clause in \mathcal{D} :

$$\frac{(P(\tilde{t}) \Leftarrow \phi' \wedge \bigwedge A') \in \mathcal{D} \quad \text{dom}(\sigma) = \text{fvs}(A') \cup \text{fvs}(\tilde{t}) \quad \vdash \phi \Rightarrow \exists \tilde{x}. (\sigma \phi') \quad \vdash \phi \Rightarrow \llbracket \sigma A' \subseteq A \rrbracket \quad \{\tilde{x}\} = \text{fvs}(\phi') \setminus \text{dom}(\sigma) \quad \mathcal{D}; \Gamma; A \cup \{P_{\circ}^{\emptyset}(\sigma \tilde{t})\}; \phi \vdash h}{\mathcal{D}; \Gamma; A; \phi \vdash h} \text{ (FOLD)}$$

Check if the current knowledge entails the asserted proposition:

$$\frac{\vdash \phi \Rightarrow \perp}{\mathcal{D}; \Gamma; A; \phi \vdash \perp} \text{ (VALID}\perp\text{)} \quad \frac{\vdash \phi \Rightarrow \llbracket P(\tilde{t}) \in A \rrbracket}{\mathcal{D}; \Gamma; A; \phi \vdash P(\tilde{t})} \text{ (VALID}P\text{)}$$

Auxiliary functions:

$$\begin{aligned} \llbracket P(\tilde{t}) \in A \rrbracket &\triangleq \bigvee_{P(\tilde{t}') \in A} \tilde{t} = \tilde{t}' \\ \llbracket \bullet \in A \rrbracket &\triangleq \top \\ \llbracket \alpha \triangleright P(\tilde{t}) \in A \rrbracket &\triangleq \llbracket P(\tilde{t}) \in \{P^M(\tilde{t}') \in A \mid \alpha \in M\} \rrbracket \\ \llbracket A_1 \subseteq A_2 \rrbracket &\triangleq \bigwedge_{P(\tilde{t}) \in A_1} \llbracket P(\tilde{t}) \in A_2 \rrbracket \end{aligned}$$

Figure 3. The inference rules for the judgment $\mathcal{D}; \Gamma; A; \phi \vdash h$.

set $\{(g_1, A_1, \phi_1, h_1), \dots, (g_m, A_m, \phi_m, h_m)\}$ consisting of user-specified lemmas and induction hypotheses, where g_i is either \bullet or $\alpha \triangleright P(\tilde{t})$. $(\bullet, A, \phi, h) \in \Gamma$ represents the user-specified lemma

$$\forall \tilde{x}. \left(\bigwedge A \wedge \phi \Rightarrow h \right) \text{ where } \{\tilde{x}\} = \text{fvs}(A, \phi, h),$$

while $(\alpha \triangleright P(\tilde{t}), A, \phi, h) \in \Gamma$ represents the induction hypothesis

$$\forall \tilde{x}. \left((P(\tilde{t}) \prec P(\tilde{t}')) \wedge \bigwedge A \wedge \phi \Rightarrow h \right) \text{ where } \{\tilde{x}\} = \text{fvs}(P(\tilde{t}), A, \phi, h)$$

that has been introduced by induction on the derivation of the atom $P(\tilde{t}')$. 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 $Q(\tilde{t})$. We will introduce a new rule **APPLY** P later in this section for using such lemmas and hypotheses to obtain new knowledge. A is also extended to be a set $\{P_{1\alpha_1}^{M_1}(t_1), \dots, P_{m\alpha_m}^{M_m}(t_m)\}$ of annotated atoms. Each element $P_{\alpha}^M(\tilde{t})$ 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 $P_{\alpha}^M(\tilde{t})$ satisfies $D \prec D'$ for the derivation D' of the atom $P(\tilde{t}')$ on which the induction with the identifier α' is performed. Thus, an induction hypothesis $(\alpha' \triangleright P(\tilde{t}'), A', \phi', h') \in \Gamma$ can be applied to the atom $P_{\alpha}^M(\tilde{t}) \in A$ only if $\alpha' \in M$ holds.

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

The inference rules for the judgment $\mathcal{D}; \Gamma; A; \phi \vdash h$ are defined in Figure 3. The rule **INDUCT** selects an atom $P_{\circ}^M(\tilde{t}) \in A$ and performs induction on the derivation of the atom. This rule generates a fresh induction identifier $\alpha \neq \circ$, adds a new induction hypothesis $(\alpha \triangleright P(\tilde{t}), A, \phi, h)$ to Γ , and replaces the atom $P_{\circ}^M(\tilde{t})$ with the annotated one $P_{\alpha}^M(\tilde{t})$ for remembering that the induction with the identifier α is performed on it. The rule **UNFOLD** selects an atom $P_{\alpha}^M(\tilde{t}) \in A$ and performs a case analysis on the last rule $P(\tilde{t}) \Leftarrow \phi_i \wedge \bigwedge A_i$ 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 $A_{i_{\circ}}^{M \cup \{\alpha\}}$ and ϕ_i respectively to A and ϕ in the i -th subgoal, where A_{α}^M represents $\{P_{\alpha}^M(\tilde{t}) \mid P(\tilde{t}) \in A\}$. 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 $P_{\alpha}^M(\tilde{t})$ 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 $P_{\alpha}^M(\tilde{t})$ yet. The rules **APPLY** \perp and **APPLY** P 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 $P(\tilde{t}) \Leftarrow \phi' \wedge \bigwedge A' \in \mathcal{D}$ 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 update it with $P(\sigma \tilde{t})$. This rule is useful when we check the correctness of user specified lemmas. The rule **VALID** \perp checks if ϕ is unsatisfiable, while the rule **VALID** P checks if the asserted proposition $P(\tilde{t})$ on the right-hand side of the judgment is implied by the current knowledge $\bigwedge A \wedge \phi$.

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 $\text{goal}(\mathcal{H}) = \{\bigwedge A_i \wedge \phi_i \Rightarrow \perp\}_{i=1}^m$, we check the judgment $\text{def}(\mathcal{H}); \emptyset; A_{i_{\circ}}^{\emptyset}; \phi_i \vdash \perp$ is derivable by the

inductive proof system. Here, each atom in A_i is initially annotated with \emptyset and \circ .

We now prove the correctness of our method, which follows from the soundness of the inductive proof system. To state the soundness, we first define $\llbracket \Gamma, A \rrbracket^k$, which represents the conjunction of user-specified lemmas and induction hypotheses in Γ instantiated for the atoms occurring in the k -times unfolding of A .

$$\begin{aligned} \llbracket \Gamma, A \rrbracket^0 &= \llbracket \Gamma \rrbracket(A), \\ \llbracket \Gamma, A \rrbracket^{k+1} &= \llbracket \Gamma \rrbracket(A) \wedge \bigwedge_{P \in \mathcal{M}(\tilde{t})} \bigvee_{i=1}^m \exists \tilde{x}_i. \left(\phi_i \wedge \llbracket \Gamma, A_{i \circ}^{M \cup \{\alpha\}} \rrbracket^k \right), \end{aligned}$$

where $\mathcal{D}(P)(\tilde{t}) = \bigvee_{i=1}^m \exists \tilde{x}_i. (\phi_i \wedge \bigwedge A_i)$ and $\llbracket \Gamma \rrbracket$ is defined by:

$$\begin{aligned} \llbracket \Gamma \rrbracket(A') &= \bigwedge_{(g, A, \phi, h) \in \Gamma} \llbracket (g, A, \phi, h) \rrbracket(A'), \\ \llbracket (\bullet, A, \phi, h) \rrbracket(A') &= \left\{ \forall \tilde{x}. \bigwedge A \wedge \phi \Rightarrow h \mid \{\tilde{x}\} = \text{fus}(A, \phi, h) \right\}, \\ \llbracket (\alpha \triangleright P(\tilde{t}), A, \phi, h) \rrbracket(A') &= \\ &\left\{ \forall \tilde{x}. \bigwedge A \wedge \phi \wedge \tilde{t} = \tilde{t}' \Rightarrow h \mid \begin{array}{l} P^M(\tilde{t}') \in A', \alpha \in M, \\ \{\tilde{x}\} = \text{fus}(\tilde{t}, A, \phi, h) \end{array} \right\}. \end{aligned}$$

Intuitively, $\llbracket \Gamma \rrbracket(A)$ represents the conjunction of user-specified lemmas and induction hypotheses in Γ instantiated for the atoms in A . The soundness of the inductive proof system is now stated by the following lemma (see Appendix A for a proof).

Lemma 1 (Soundness). *If $\mathcal{D}; \Gamma; A; \phi \vdash h$ is derivable, then there is k such that $\mu F_{\mathcal{D}} \models \llbracket \Gamma, A \rrbracket^k \wedge \bigwedge A \wedge \phi \Rightarrow h$ holds.*

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

Theorem 2. *Suppose that \mathcal{H} is an HCCS with $\text{goal}(\mathcal{H}) = \{\bigwedge A_i \wedge \phi_i \Rightarrow \perp\}_{i=1}^m$. It then follows that $\rho \models \mathcal{H}$ for some ρ if $\text{def}(\mathcal{H}); \emptyset; A_i; \phi_i \vdash \perp$ is derivable for all $i = 1, \dots, m$.*

Proof. Suppose that $\text{def}(\mathcal{H}); \emptyset; A_i; \phi_i \vdash \perp$ for all $i = 1, \dots, m$. By Lemma 1 and the fact that $\mu F_{\text{def}(\mathcal{H})} \models \bigwedge A_i \Rightarrow \llbracket \emptyset, A_i \rrbracket^k$, we get $\mu F_{\text{def}(\mathcal{H})} \models \bigwedge A_i \wedge \phi_i \Rightarrow \perp$. We therefore have $\mu F_{\text{def}(\mathcal{H})} \models \text{goal}(\mathcal{H})$. It then follows that $\rho \models \mathcal{H}$ for some ρ by Corollary 1. \square

4.2 Rule Application Strategy

We now elaborate on our rule application strategy shown in Section 2. Because all the inference rules except $\text{VALID}\perp$ and $\text{VALID}P$ add new knowledge to A and/or ϕ , we repeatedly apply them until $\text{VALID}\perp$ and $\text{VALID}P$ close all the proof branches under consideration. More specifically, we adopt the following strategy:

- Repeatedly apply the rules $\text{APPLY}\perp$, $\text{APPLY}P$, and FOLD if possible until no new knowledge is obtained. (Even if the rules do not apply, applications of INDUCT and UNFOLD explained in the following items may make $\text{APPLY}\perp$, $\text{APPLY}P$, and FOLD applicable.)
- If the current knowledge cannot be updated by using the above rules, 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 rules $\text{VALID}\perp$ and $\text{VALID}P$ whenever the current knowledge is updated.

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 an existing verification tool called Refinement Caml [52–54], a refinement type checking and inference tool for the OCaml functional language based on Horn constraint solving. Our solver can generate a proof tree like the one in Figure 2 as a certificate, if the given Horn constraint set is judged to have a solution. Furthermore, our solver can generate a counterexample, if the constraint set is judged to be unsolvable. We adopted Z3 [19] as the underlying SMT solver. The details of the implementation are explained in Section 5.1. The web interface of the verification tool as well as all the benchmark programs used in the experiments reported here are available from <http://www.cs.tsukuba.ac.jp/~uhiro/>.

We have tested our constraint solver on two benchmark sets. The first set is 85 benchmarks from the test suite for automated induction provided by the authors of the IsaPlanner system [21]. The benchmark set consists of verification problems of relational specifications of pure mathematical functions on inductive data structures, most of which cannot be verified by the previous Horn constraint solvers [25, 26, 31, 41, 46, 50, 53, 54]. The benchmark set has also been used to evaluate previous automated inductive theorem provers [15, 40, 44, 48]. The experiment results on this benchmark set are reported in Section 5.2.

To demonstrate advantages of our novel combination of Horn constraint solving with inductive theorem proving, we have prepared the second benchmark set consisting of verification problems of (mostly relational) specifications of programs that use various advanced language features, which are naturally and automatically axiomatized by our method using predicates defined by Horn clause constraints as the least satisfying interpretation. The experiment results on this benchmark set are reported in Section 5.3.

5.1 Implementation Details

This section describes details of the implementation. We explain how to check the correctness of user specified lemmas and how to generate a counterexample if the given Horn constraint set has no solution, respectively in Sections 5.1.1 and 5.1.2. Section 5.1.3 describes implementation details of the rules $\text{APPLY}\perp$, $\text{APPLY}P$, and FOLD in Figure 3. In particular, we discuss how to find an assignment σ for free term variables that occur in the element of Γ selected by the rules.

5.1.1 Checking the correctness of user-specified lemmas

Our system allows users to specify lemmas as the initial Γ . Our tool checks that $\mathcal{D}; \emptyset; A; \phi \vdash h$ is derivable for each user-specified lemma (\bullet, A, ϕ, h) by using the exact same rules in Figure 3. We use the rule FOLD , in addition to the rules $\text{APPLY}\perp$ and $\text{APPLY}P$, to update the current knowledge. To avoid redundant applications of FOLD , we select only definite clauses in \mathcal{D} with the head of the form $P(\tilde{t}')$ if $h = P(\tilde{t})$ and we do not use FOLD at all if $h = \perp$.

5.1.2 Counterexample generation

Our tool can conclude that the goal clause (or the user-specified lemma) currently solving has no solution if a subgoal of the form $\mathcal{D}; \Gamma; A; \phi \vdash \perp$ satisfying the following conditions is obtained:

- all the atoms in A are already unfolded by the rule UNFOLD but
- ϕ is satisfiable.

Note that the first condition ensures that the ϕ -part of the current knowledge under-approximates the body of the goal clause.

Our tool then returns a satisfying model of ϕ found by the underlying SMT solver as a counterexample witnessing the unsolvability of the given Horn constraint set. Some readers may notice that the

counterexample generation is essentially the same as the execution of constraint logic programs [33].

5.1.3 Finding an assignment σ for quantifier instantiation

We here explain how to find σ for instantiating quantified variables of lemmas and induction hypotheses by using an SMT solver in the implementation of the rule $\text{APPLY}\perp$. The same technique is also used for finding σ in the rules FOLD and $\text{APPLY}P$.

Recall that, in order to apply the rule $\text{APPLY}\perp$ to a judgment $\mathcal{D}; \Gamma; A; \phi \vdash h$, we need to find an assignment σ for free term variables in $(g, A', \phi', h') \in \Gamma$. Note here that the atom that occurs in g also occurs in A' . We below assume that all the arguments of the atoms in A' are distinct term variables. This does not lose generality because we can always replace $P(\tilde{t}) \in A'$ by $P(\tilde{x})$ with fresh \tilde{x} by adding the constraint $\tilde{x} = \tilde{t}$ to ϕ' . First of all, we construct, for each $P(\tilde{x}) \in A'$, the set $\{\{\tilde{x} \mapsto \tilde{t}_i\}\}_{i=1}^m$ of assignments, where $P(\tilde{t}_1), \dots, P(\tilde{t}_m) \in A$. Here, if $g = \alpha \triangleright P(\tilde{x})$, the set $\{\{\tilde{x} \mapsto \tilde{t}_i\}\}_{i=1}^m$ of assignments is constructed only from $P^{M_1}(\tilde{t}_1), \dots, P^{M_m}(\tilde{t}_m) \in A$ such that $\alpha \in M_i$. For example, let us consider $g = \alpha \triangleright P_1(\tilde{x}_1)$, $A' = \{P_1(\tilde{x}_1), P_1(\tilde{x}_2), P_2(\tilde{x}_3)\}$ and $A = \{P_1^{\{\alpha\}}(\tilde{t}_1), P_1^{\emptyset}(\tilde{t}_2), P_2^{\emptyset}(\tilde{t}_3)\}$. For the atoms $P_1(\tilde{x}_1)$, $P_1(\tilde{x}_2)$, and $P_2(\tilde{x}_3)$, we respectively obtain the sets $\{\{\tilde{x}_1 \mapsto \tilde{t}_1\}\}$, $\{\{\tilde{x}_2 \mapsto \tilde{t}_1\}, \{\tilde{x}_2 \mapsto \tilde{t}_2\}\}$, and $\{\{\tilde{x}_3 \mapsto \tilde{t}_3\}\}$ of assignments. We then compute all the combination of assignments, and filter out those that contradict with ϕ . For the above example, we obtain the following two as candidates of σ :

$$\begin{aligned}\sigma_1 &= \{\tilde{x}_1 \mapsto \tilde{t}_1, \tilde{x}_2 \mapsto \tilde{t}_1, \tilde{x}_3 \mapsto \tilde{t}_3\} \\ \sigma_2 &= \{\tilde{x}_1 \mapsto \tilde{t}_1, \tilde{x}_2 \mapsto \tilde{t}_2, \tilde{x}_3 \mapsto \tilde{t}_3\}\end{aligned}$$

For the rules FOLD and $\text{APPLY}P$, we use the same technique explained above for $\text{APPLY}\perp$, but additionally check the condition $\models \phi \Rightarrow \exists \tilde{x}. \sigma\phi'$, where $\{\tilde{x}\} = \text{fvs}(\phi') \setminus \text{dom}(\sigma)$.

5.2 Experiments on IsaPlanner benchmark set

The IsaPlanner benchmark set consists of 85 conjectures for total recursive functions on inductively defined data structures such as natural numbers, lists, and binary trees. We have translated these conjectures into assertion safety verification problems of OCaml programs. In the translation, we encoded natural numbers using integer primitives, and defined lists and binary trees as algebraic data types in OCaml. More specifically, natural numbers Z and $S\ t$ are respectively encoded as 0 and $t' + 1$ for t' obtained by encoding t . To preserve the semantics of natural numbers, we translated conjectures of the form $\forall x \in \mathbb{N}. \phi$ into $\forall x \in \mathbb{Z}. (x \geq 0 \Rightarrow \phi)$.

The translated verification problems are then verified by our verification tool. Our tool automatically reduced the verification problems into Horn constraint solving problems by using the constraint generation method [53], and automatically (i.e., without using user-specified lemmas) solved 68 out of 85 verification problems. We have manually analyzed the experiment results and found that 8 out of 17 failed verification problems require lemma discovery. The other 9 problems caused timeout of Z3. It was because the rule application strategy implemented in our tool 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 cause significant negative impacts on the automation of induction; According to [48] that uses the IsaPlanner benchmark set to compare state-of-the-art automated inductive theorem provers based on logics of pure total functions over inductively-defined data structures, IsaPlanner [21] proved 47 out of 85, Dafny [40] proved 45, ACL2s [13] proved 74, and Zeno [48] proved 82. The HipSpec [15] inductive prover and the SMT solver CVC4

extended with induction [44] 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 inductively-defined data structures.

It is also worth noting that, all the inductive provers that won best results (greater than 70) on the benchmark set support automatic lemma discovery, in a stark contrast to our tool. For example, the above result (80 out of 85) of CVC4 is obtained when they enable an automatic lemma discovery technique proposed in [44] and use a different encoding (called **dti** in [44]) of natural numbers than ours. When they disable the lemma discovery technique and use a similar encoding to ours (called **dtf** in [44]), 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 [13, 15, 32, 36, 44, 48], further makes induction-based Horn constraint solving powerful.

5.3 Experiments on benchmark set consisting of programs with various advanced language features

We prepared and tested our tool with the second benchmark set consisting of (mostly relational) assertion safety verification problems of programs that use various advanced language features such as partial (i.e., possibly non-terminating) functions, higher-order functions, exceptions, non-determinism, algebraic data types, 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 four (ID19–22 in Table 1) are relational ones where safe inductive invariants are not expressible in $\text{QF}\text{-LIA}$, and therefore not solvable by the previous Horn constraint solvers. As shown in Section 3.3, these verification problems are naturally and automatically axiomatized by our method using predicates defined by Horn clause constraints as the least satisfying interpretation. By contrast, these assertion safety verification problems cannot be straightforwardly axiomatized and proved by the previous automated inductive theorem provers based on logics of pure total functions on inductively-defined data structures: the axiomatization process of these verification problems using pure total functions often requires users' manual intervention and possibly causes a negative effect on the automation of induction, because, in the process, 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. Additionally, the axiomatization process needs to preserve branching and calling context information in order to perform path- and context-sensitive verification.

Table 1 summarizes the experiment results on the benchmark set. The column “specification” represents the relational specification verified and the column “kind” shows the kind of the specification, 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 “language 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

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

ID	specification	kind	language features	result	time (sec.)
1	<code>mult x y + a = mult_acc x y a</code>	equiv	P	✓	0.257
2	<code>mult x y = mult_acc x y 0</code>	equiv	P	✓ [†]	0.435
3	<code>mult (1 + x) y = y + mult x y</code>	equiv	P	✓	0.233
4	<code>y ≥ 0 ⇒ mult x (1 + y) = x + mult x y</code>	equiv	P	✓	0.248
5	<code>mult x y = mult y x</code>	comm	P	✓ [‡]	0.345
6	<code>mult (x + y) z = mult x z + mult y z</code>	dist	P	✓	1.276
7	<code>mult x (y + z) = mult x y + mult x z</code>	dist	P	✗	n/a
8	<code>mult (mult x y) z = mult x (mult y z)</code>	assoc	P	✗	n/a
9	<code>0 ≤ x₁ ≤ x₂ ∧ 0 ≤ y₁ ≤ y₂ ⇒ mult x₁ y₁ ≤ mult x₂ y₂</code>	mono	P	✓	0.265
10	<code>sum x + a = sum_acc x a</code>	equiv		✓	0.384
11	<code>sum x = x + sum (x - 1)</code>	equiv		✓	0.272
12	<code>x ≤ y ⇒ sum x ≤ sum y</code>	mono		✓	0.350
13	<code>x ≥ 0 ⇒ sum x = sum_down 0 x</code>	equiv	P	✓	0.312
14	<code>x < 0 ⇒ sum x = sum_up x 0</code>	equiv	P	✓	0.368
15	<code>sum_down x y = sum_up x y</code>	equiv	P	✗	n/a
16	<code>sum x = apply sum x</code>	equiv	H	✓	0.286
17	<code>mult x y = apply2 mult x y</code>	equiv	H, P	✓	0.279
18	<code>repeat x (add x) a y = a + mult x y</code>	equiv	H, P	✓	0.317
19	<code>x ≤ 101 ⇒ mc91 x = 91</code>	nonrel	I	✓	0.165
20	<code>x ≥ 0 ∧ y ≥ 0 ⇒ ack x y > y</code>	nonrel	I	✓	0.212
21	<code>x ≥ 0 ⇒ 2 × sum x = x × (x + 1)</code>	nonrel	N	✓	0.196
22	<code>dyn_sys 0. - / → * assert false</code>	nonrel	R, N	✓	0.144
23	<code>flip_mod y x = flip_mod y (flip_mod y x)</code>	idem	P	✓	7.712
24	<code>noninter h₁ l₁ l₂ l₃ = noninter h₂ l₁ l₂ l₃</code>	nonint	P	✓	0.662
25	<code>try find_opt p l = Some (find p l) with Not_Found → find_opt p l = None</code>	equiv	H, E	✓	0.758
26	<code>try mem (find ((=) x) l) l with Not_Found → ¬(mem x l)</code>	equiv	H, E	✓	0.764
27	<code>sum_list l = fold_left (+) 0 l</code>	equiv	H	✓	3.681
28	<code>sum_list l = fold_right (+) l 0</code>	equiv	H	✓	0.329
29	<code>sum_fun randpos n > 0</code>	equiv	H, D	✓	0.240
30	<code>mult x y = mult_Ccode(x, y)</code>	equiv	P, C	✓	0.217

[†] A lemma $P_{\text{mult_acc}}(x, y, a, r) \Rightarrow P_{\text{mult_acc}}(x, y, a - x, r - x)$ is used

[‡] A lemma $P_{\text{mult}}(x, y, r) \Rightarrow P_{\text{mult}}(x - 1, y, r - y)$ is used

P_f above represents the predicate that axiomatizes the function f .

The experiments were conducted on a machine with Intel(R) Xeon(R) CPU E5-2680 v3 (2.50 GHz, 16 GB of memory).

N: nonlinear functions

C: procedures written in different programming paradigms

The column “result” represents whether our verification method succeeded ✓ or failed ✗. The column “time” represents the elapsed time for verification in seconds.

Overall, the experiment results are promising, which show that our tool can automatically 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 recurses on y . Thus, the result demonstrates an advantage of our induction-based method that it can exploit lemmas to fill the gap between function calls with different control flows. Our tool, however, failed to verify the distributivity ID7 of `mult`, the associativity ID8 of `mult`, and the equivalence ID15 of `sum_down` and `sum_up`. ID7 could be reduced to ID6 and solved, if a lemma $P_{\text{mult}}(x, y, r) \Rightarrow P_{\text{mult}}(y, x, r)$, which represents the commutativity of `mult`, was used to rewrite the conjecture

$$P_{\text{mult}}(x, y+z, s_1) \wedge P_{\text{mult}}(x, y, s_2) \wedge P_{\text{mult}}(x, z, s_3) \Rightarrow s_1 = s_2 + s_3$$

obtained from the specification `mult x (y + z) = mult x y + mult x z` into

$$P_{\text{mult}}(y+z, x, s_1) \wedge P_{\text{mult}}(y, x, s_2) \wedge P_{\text{mult}}(z, x, s_3) \Rightarrow s_1 = s_2 + s_3$$

by replacing atoms of the form $P_{\text{mult}}(t_1, t_2, t_3)$ with $P_{\text{mult}}(t_2, t_1, t_3)$. The rule `APPLYP`, however, replaces each atom $P_{\text{mult}}(t_1, t_2, t_3)$ with $P_{\text{mult}}(t_1, t_2, t_3) \wedge P_{\text{mult}}(t_2, t_1, t_3)$ instead by keeping the original atom so that we can monotonically increase the current knowledge. Our tool supports an option for the rule `APPLYP` of eliminating the original atom, and if it is enabled, ID7 is verified. The associativity verification problem ID8 is even more difficult. In addition to the above lemma, a lemma $P_{\text{mult}}(x + y, z, r) \Rightarrow \exists s_1, s_2. (P_{\text{mult}}(x, z, s_1) \wedge P_{\text{mult}}(y, z, s_2) \wedge r = s_1 + s_2)$ is required. This lemma, however, is currently not of the form supported by our inductive proof system. In ID15, the functions `sum_down` and `sum_up` use different recursion parameters (resp. y and x), and requires lemmas $P_{\text{sum_down}}(x, y, s) \Rightarrow \exists s_1, s_2. (P_{\text{sum_down}}(0, y, s_1) \wedge P_{\text{sum_down}}(0, x - 1, s_2) \wedge s = s_1 - s_2)$ and $P_{\text{sum_up}}(x, y, s) \Rightarrow \exists s_1, s_2. (P_{\text{sum_down}}(0, y, s_1) \wedge P_{\text{sum_down}}(0, x - 1, s_2) \wedge s = s_1 - s_2)$. These lemmas are provable by induction on the derivation of $P_{\text{sum_down}}(x, y, s)$ and $P_{\text{sum_up}}(x, y, s)$, respectively. However, as in the case of ID8, our proof system does not support the form of the lemmas. To put it differently, ID8 and ID15 demonstrate the in-

completeness of our inductive proof system. Our future work thus includes an extension of the proof system to support more general form of lemmas and judgments.

6. Related Work

As discussed in Section 1, Horn constraint solving methods have been extensively studied [25, 26, 31, 41, 46, 50, 53, 54]. In contrast to the proposed induction based method, these methods do not support Horn clause constraints over the theories of algebraic data structures and nonlinear arithmetics, and cannot verify most if not all relational specifications shown in Section 5.

Because state-of-the-art SMT solvers such as Z3 [19] and CVC4 support quantifier instantiation heuristics, one may think that they alone are sufficient for checking the validity of the logical interpretation of Horn clause constraints shown in Section 3.1. However, they alone are not sufficient for proving most conjectures that require nontrivial use of induction such as the benchmark problems in Section 5.³ In fact, [44] reports that Z3 (resp. CVC4 without induction) alone have proved only 35 (resp. 34) out of 85 problems in the IsaPlanner benchmark set.

Automated inductive theorem proving techniques and tools have long been studied, for example and to name a few: the Boyer-Moore theorem provers [36] like ACL2s [13], rewriting induction provers [43] like SPIKE [9], proof planners like CLAM [11, 12, 32, 34] and IsaPlanner [20, 21], and SMT-based induction provers like Leon [49], Dafny [40], Zeno [48], HipSpec [15], and CVC4 [44]. 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 (e.g., ones discussed in Section 3.3) 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 [49] 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 such as Hoare logics and refinement type systems.

To aid verification of relational specifications of functional programs, Giesl [23] proposed context-moving transformations and Asada et al. [1] proposed a kind of tupling transformation. SymDiff [30, 39] is a transformation-based tool built on top of Boogie [2] for equivalence verification of imperative programs. Self-composition [3] is a program transformation technique to reduce k-safety [16, 51] verification into ordinary safety verification, and has been applied to non-interference [4, 51, 55] and regression verification [22] of imperative programs. These transformations are useful for some patterns of relational verification problems, which are, however, less flexible 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 in Table 1), but does not help verification of the commutativity of `mult` (ID5 in Table 1). Because each transformation is designed for a particular target language, the transformations cannot be applied to aid relational verification across programs written in different paradigms (e.g., ID30 in Table 1). Moreover, the correctness proof of the transformations tends to be harder because it involves the operational semantics of

the target language, which is complex compared to the logical semantics of Horn clause constraints.

There have been proposed program logics for relational verification [5, 6, 14, 24]. 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 is, however, not automated.

7. Conclusion and Future Work

We have proposed a novel Horn constraint solving method based on an inductive proof system and an SMT-based technique to automate proof search in the system. We have shown that our method is able to solve Horn clause constraints obtained from relational verification problems that were not possible with the previous methods based on interpolating theorem proving. 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 [13, 15, 44, 48]. 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 [30] across multiple predicates (e.g., P, Q of \mathcal{H}_{mult}).

References

- [1] 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.
- [2] 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.
- [3] G. Barthe, P. R. D’Argenio, and T. Rezk. Secure information flow by self-composition. In *CSFW ’04*, pages 100–114. IEEE, 2004.
- [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, B. Köpf, F. Olmedo, and S. Zanella Béguelin. Probabilistic relational reasoning for differential privacy. In *POPL ’12*, pages 97–110. ACM, 2012.
- [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] T. A. Beyene, C. Popeea, and A. Rybalchenko. Solving existentially quantified horn clauses. In *CAV ’13*, volume 8044 of *LNCS*, pages 869–882. Springer, 2013.
- [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] A. Bundy. The automation of proof by mathematical induction. In *Handbook of Automated Reasoning*, volume I, pages 845–911. Elsevier, 2001.

³This point is also mentioned in the tutorial of Z3 (<http://rise4fun.com/Z3/tutorial/guide>).

- [12] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The oyster-clam system. In *CADE-10*, pages 647–648. Springer, 1990.
- [13] H. R. Chamathi, 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.
- [14] Ș. Ciobăcă, D. Lucanu, V. Rusu, and G. Roșu. A language-independent proof system for mutual program equivalence. In *ICFEM '14*, pages 75–90. Springer, 2014.
- [15] 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.
- [16] M. R. Clarkson and F. B. Schneider. Hyperproperties. In *CSF '08*, pages 51–65. IEEE, 2008.
- [17] 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.
- [18] W. Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Logic*, 22:269–285, 1957.
- [19] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS '08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
- [20] L. Dixon and J. Fleuriot. Higher order rippling in IsaPlanner. In *TPHOLs '04*, pages 83–98. Springer, 2004.
- [21] 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.
- [22] D. Felsing, S. Grebing, V. Klebanov, P. Rümmer, and M. Ulbrich. Automating regression verification. In *ASE '14*, pages 349–360. ACM, 2014.
- [23] J. Giesl. Context-moving transformations for function verification. In *LOPSTR '00*, volume 1817 of *LNCS*, pages 293–312. Springer, 2000.
- [24] B. Godlin and O. Strichman. Regression verification: proving the equivalence of similar programs. *Software Testing, Verification and Reliability*, 23(3):241–258, 2013.
- [25] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI '12*, pages 405–416. ACM, 2012.
- [26] 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.
- [27] A. Gupta, C. Popeea, and A. Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In *POPL '11*, pages 331–344. ACM, 2011.
- [28] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The SeaHorn verification framework. In *CAV '15*, pages 343–361. Springer, 2015.
- [29] K. Hashimoto and H. Unno. Refinement type inference via horn constraint optimization. In *SAS '15*, volume 9291 of *LNCS*, pages 199–216. Springer, 2015.
- [30] 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.
- [31] 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.
- [32] A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1-2):79–111, 1996.
- [33] J. Jaffar and M. J. Maher. Constraint logic programming: a survey. *The Journal of Logic Programming*, 19:503 – 581, 1994.
- [34] M. Johansson, L. Dixon, and A. Bundy. Case-analysis for rippling and inductive proof. In *ITP '10*, volume 6172 of *LNCS*, pages 291–306. Springer, 2010.
- [35] T. Kahsai, P. Rümmer, H. Sanchez, and M. Schäf. JayHorn: A framework for verifying Java programs. In *CAV '16*. Springer, 2016.
- [36] M. Kaufmann, J. S. Moore, and P. Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.
- [37] K. Knowles and C. Flanagan. Type reconstruction for general refinement types. In *ESOP '07*, volume 4421 of *LNCS*, pages 505–519. Springer, 2007.
- [38] S. Kupferschmid and B. Becker. Craig interpolation in the presence of non-linear constraints. In *FORMATS '11*, pages 240–255. Springer, 2011.
- [39] 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.
- [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, 2013.
- [42] K. L. McMillan. An interpolating theorem prover. *Theoretical Computer Science*, 345(1):101–121, 2005.
- [43] U. S. Reddy. Term rewriting induction. In *CADE-10*, volume 449 of *LNCS*, pages 162–177. Springer, 1990.
- [44] A. Reynolds and V. Kuncak. Induction for SMT solvers. In *VMCAI '15*, volume 8931 of *LNCS*, pages 80–98. Springer, 2015.
- [45] P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI '08*, pages 159–169. ACM, 2008.
- [46] 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.
- [47] R. Sato, H. Unno, and N. Kobayashi. Towards a scalable software model checker for higher-order programs. In *PEPM '13*, pages 53–62. ACM, 2013.
- [48] 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, March 2012.
- [49] P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *SAS '11*, volume 6887 of *LNCS*, pages 298–315. Springer, 2011.
- [50] T. Terauchi. Dependent types from counterexamples. In *POPL '10*, pages 119–130. ACM, 2010.
- [51] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *SAS '05*, volume 3672 of *LNCS*, pages 352–367. Springer, 2005.
- [52] H. Unno and N. Kobayashi. On-demand refinement of dependent types. In *FLOPS '08*, volume 4989 of *LNCS*, pages 81–96. Springer, 2008.
- [53] H. Unno and N. Kobayashi. Dependent type inference with interpolants. In *PPDP '09*, pages 277–288. ACM, 2009.
- [54] 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.
- [55] 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.
- [56] 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.
- [57] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL '99*, pages 214–227. ACM, 1999.

A. Proof of Lemma 1

We first show lemmas used to prove Lemma 1.

Lemma 2. $\models \phi \Rightarrow \llbracket P(\tilde{t}) \in A \rrbracket$ implies $\models \bigwedge A \wedge \phi \Rightarrow P(\tilde{t})$.

Proof. By the definition of $\llbracket P(\tilde{t}) \in A \rrbracket$. \square

Lemma 3. $\models \phi \Rightarrow \llbracket A_1 \subseteq A_2 \rrbracket$ implies $\models \bigwedge A_2 \wedge \phi \Rightarrow \bigwedge A_1$.

Proof. By the definition of $\llbracket A_1 \subseteq A_2 \rrbracket$ and Lemma 2. \square

Lemma 4. Suppose that $(g, A', \phi', h) \in \Gamma$ and $\models \phi \Rightarrow \llbracket \sigma g \in A \rrbracket$ for some σ with $\text{dom}(\sigma) = \text{fvs}(A') \cup \text{fvs}(h)$. It then follows that $\models \llbracket \Gamma \rrbracket(A) \wedge \bigwedge \sigma A' \wedge \phi \Rightarrow (\sigma h \vee \forall \tilde{x}. \neg(\sigma \phi'))$, where $\{\tilde{x}\} = \text{fvs}(\phi') \setminus \text{dom}(\sigma)$.

Proof. We perform a case analysis on g .

- Suppose that $g = \bullet$. By the definition of $\llbracket \Gamma \rrbracket(A)$, we obtain

$$\models \llbracket \Gamma \rrbracket(A) \Rightarrow \forall \tilde{x}'. \left(\bigwedge A' \wedge \phi' \Rightarrow h \right),$$

where $\{\tilde{x}'\} = \text{fvs}(A') \cup \text{fvs}(\phi') \cup \text{fvs}(h)$. Therefore, we get

$$\models \llbracket \Gamma \rrbracket(A) \Rightarrow \bigwedge \sigma A' \wedge \exists \tilde{x}. (\sigma \phi') \Rightarrow \sigma h,$$

where $\{\tilde{x}\} = \text{fvs}(\phi') \setminus \text{dom}(\sigma)$. It immediately follows that

$$\models \llbracket \Gamma \rrbracket(A) \wedge \bigwedge \sigma A' \wedge \phi \Rightarrow (\sigma h \vee \forall \tilde{x}. \neg(\sigma \phi')).$$

- Suppose that $g = \alpha \triangleright P(\tilde{t})$. We assume that ϕ is not equivalent to \perp (otherwise, the lemma is trivial). By $\models \phi \Rightarrow \llbracket \sigma g \in A \rrbracket$, there is $P^M(\tilde{t}') \in A$ such that $\alpha \in M$ and $\models \phi \Rightarrow \sigma \tilde{t} = \tilde{t}'$. By the definition of $\llbracket \Gamma \rrbracket(A)$, we obtain

$$\models \llbracket \Gamma \rrbracket(A) \Rightarrow \forall \tilde{x}'. \left(\bigwedge A' \wedge \phi' \wedge \tilde{t} = \tilde{t}' \Rightarrow h \right),$$

where $\{\tilde{x}'\} = \text{fvs}(\tilde{t}) \cup \text{fvs}(A') \cup \text{fvs}(\phi') \cup \text{fvs}(h)$. Therefore, we get

$$\models \llbracket \Gamma \rrbracket(A) \Rightarrow \bigwedge \sigma A' \wedge \exists \tilde{x}. (\sigma \phi') \wedge \sigma \tilde{t} = \tilde{t}' \Rightarrow \sigma h,$$

where $\{\tilde{x}\} = \text{fvs}(\phi') \setminus \text{dom}(\sigma)$. We thus obtain

$$\models \llbracket \Gamma \rrbracket(A) \wedge \bigwedge \sigma A' \wedge \phi \Rightarrow (\sigma h \vee \forall \tilde{x}. \neg(\sigma \phi')).$$

\square

Lemma 5. if $k_1 \leq k_2$, then $\models \llbracket \Gamma, A \rrbracket^{k_2} \Rightarrow \llbracket \Gamma, A \rrbracket^{k_1}$ holds.

Proof. By the definition of $\llbracket \bullet, \bullet \rrbracket^\bullet$. \square

Lemma 6. We have

$$\mu F_{\mathcal{D}} \models \llbracket \Gamma, A \rrbracket^k \wedge P(\tilde{t}) \Rightarrow \llbracket \Gamma, A \cup \{P^0(\tilde{t})\} \rrbracket^k.$$

Proof. By the definition of $\llbracket \bullet, \bullet \rrbracket^\bullet$. \square

Lemma 7. If $P_\alpha^M(\tilde{t}) \in A$, then for each $i \in \{1, \dots, m\}$,

$$\mu F_{\mathcal{D}} \models \llbracket \Gamma, A \rrbracket^{k+1} \Rightarrow \bigvee_{i=1}^m \exists \tilde{x}_i. \left(\phi_i \wedge \llbracket \Gamma, A \cup A_{i_0}^{M \cup \{\alpha\}} \rrbracket^k \right),$$

where $\mathcal{D}(P)(\tilde{t}) = \bigvee_{i=1}^m \exists \tilde{x}_i. (\phi_i \wedge \bigwedge A_i)$.

Proof. By the definition of $\llbracket \bullet, \bullet \rrbracket^\bullet$. \square

Lemma 8. Suppose that $P_\alpha^M(\tilde{t}) \in A$ and α does not occur in Γ or A . We then obtain

$$\models \llbracket \Gamma, A \rrbracket^k \wedge \llbracket \{(\alpha \triangleright P(\tilde{t}), A, \phi, h)\}, \{P_\alpha^M(\tilde{t})\} \rrbracket^k \Rightarrow \llbracket \Gamma \cup \{(\alpha \triangleright P(\tilde{t}), A, \phi, h)\}, (A \setminus P_\alpha^M(\tilde{t})) \cup \{P_\alpha^M(\tilde{t})\} \rrbracket^k$$

Proof. By the definition of $\llbracket \bullet, \bullet \rrbracket^\bullet$, we get

$$\models \llbracket \Gamma, A \rrbracket^k \wedge \llbracket \{(\alpha \triangleright P(\tilde{t}), A, \phi, h)\}, A \rrbracket^k \wedge \llbracket \Gamma, \{P_\alpha^M(\tilde{t})\} \rrbracket^k \wedge \llbracket \{(\alpha \triangleright P(\tilde{t}), A, \phi, h)\}, \{P_\alpha^M(\tilde{t})\} \rrbracket^k \Rightarrow \llbracket \Gamma \cup \{(\alpha \triangleright P(\tilde{t}), A, \phi, h)\}, (A \setminus P_\alpha^M(\tilde{t})) \cup \{P_\alpha^M(\tilde{t})\} \rrbracket^k$$

Because α does not occur in Γ or A and $P_\alpha^M(\tilde{t}) \in A$, we obtain

$$\models \llbracket \Gamma, A \rrbracket^k \wedge \llbracket \{(\alpha \triangleright P(\tilde{t}), A, \phi, h)\}, \{P_\alpha^M(\tilde{t})\} \rrbracket^k \Rightarrow \llbracket \Gamma \cup \{(\alpha \triangleright P(\tilde{t}), A, \phi, h)\}, (A \setminus P_\alpha^M(\tilde{t})) \cup \{P_\alpha^M(\tilde{t})\} \rrbracket^k$$

\square

Because $\mu F_{\mathcal{D}}$ is the least interpretation, we obtain:

Lemma 9. For all $P \in \text{pvs}(\mathcal{D})$ and ψ with $\text{fvs}(\psi) \subseteq \{\tilde{x}\}$, we get

$$\mu F_{\mathcal{D}} \models \forall \tilde{x}. (\{P \mapsto \lambda \tilde{x}. \psi\} \mathcal{D}(P)(\tilde{x}) \Rightarrow \psi) \Rightarrow \forall \tilde{x}. (P(\tilde{x}) \Rightarrow \psi)$$

Lemma 10. For all $P \in \text{pvs}(\mathcal{D})$ and φ with $\text{fvs}(\varphi) \subseteq \{\tilde{x}\}$, we get

$$\mu F_{\mathcal{D}} \models \forall \tilde{x}. ((\forall p \prec P. \{P \mapsto \lambda \tilde{x}. \varphi\} p(\tilde{x})) \Rightarrow \varphi) \Rightarrow \forall \tilde{x}. (P(\tilde{x}) \Rightarrow \varphi),$$

where $p \prec P$ is a predicate obtained by unfolding P at least once.

Proof. By Lemma 9 with $\psi = \forall p \prec P. \{P \mapsto \lambda \tilde{x}. \varphi\} p(\tilde{x})$. \square

Lemma 11. Suppose that $P \in \text{pvs}(\mathcal{D})$, $P_\alpha^M(\tilde{t}) \in A$, α does not occur in A , and the following holds

$$\mu F_{\mathcal{D}} \models \llbracket \{(\alpha \triangleright P(\tilde{t}), A, \phi, h)\}, \{P_\alpha^M(\tilde{t})\} \rrbracket^k \wedge \bigwedge A \wedge \phi \Rightarrow h$$

It then follows that $\mu F_{\mathcal{D}} \models \bigwedge A \wedge \phi \Rightarrow h$.

Proof. By Lemma 10. \square

We now prove Lemma 1.

Proof of Lemma 1. By induction on the derivation of $\mathcal{D}; \Gamma; A; \phi \vdash h$.

- **Case INDUCT:** We have

- $P_\alpha^M(\tilde{t}) \in A$,
- $\Gamma' = \Gamma \cup \{(\alpha \triangleright P(\tilde{t}), A, \phi, h)\}$,
- $\mathcal{D}; \Gamma'; A'; \phi \vdash h$,
- $A' = (A \setminus P_\alpha^M(\tilde{t})) \cup \{P_\alpha^M(\tilde{t})\}$, and
- α is fresh.

By I.H., there is k' such that

$$\mu F_{\mathcal{D}} \models \llbracket \Gamma', A' \rrbracket^{k'} \wedge \bigwedge A' \wedge \phi \Rightarrow h$$

By Lemma 8, we get

$$\mu F_{\mathcal{D}} \models \llbracket \{(\alpha \triangleright P(\tilde{t}), A, \phi, h)\}, \{P_\alpha^M(\tilde{t})\} \rrbracket^{k'} \wedge \bigwedge A' \wedge \phi \wedge \llbracket \Gamma, A \rrbracket^{k'} \Rightarrow h$$

By the fact $\models A \Leftrightarrow A'$ and the definition of $\llbracket \bullet, \bullet \rrbracket^\bullet$, we get

$$\mu F_{\mathcal{D}} \models \left[\left\{ (\alpha \triangleright P(\tilde{t}), A, \phi \wedge \llbracket \Gamma, A \rrbracket^{k'}, h) \right\}, \left\{ P_{\alpha}^M(\tilde{t}) \right\} \right]^{k'} \wedge \bigwedge A \wedge \phi \wedge \llbracket \Gamma, A \rrbracket^{k'} \Rightarrow h$$

Therefore, by Lemma 11, for $k = k'$, we get

$$\mu F_{\mathcal{D}} \models \llbracket \Gamma, A \rrbracket^k \wedge \bigwedge A \wedge \phi \Rightarrow h$$

• **Case UNFOLD:** For each $i \in \{1, \dots, m\}$, we have

- $P_{\alpha}^M(\tilde{t}) \in A$,
- $\mathcal{D}(P)(\tilde{t}) = \bigvee_{i=1}^m \exists \tilde{x}_i. (\phi_i \wedge \bigwedge A_i)$, and
- $\mathcal{D}; \Gamma; A \cup A_{i_0}^{M \cup \{\alpha\}}; \phi \wedge \phi_i \vdash h$.

By I.H., for each $i \in \{1, \dots, m\}$, there is k_i such that

$$\mu F_{\mathcal{D}} \models \left[\Gamma, A \cup A_{i_0}^{M \cup \{\alpha\}} \right]^{k_i} \wedge \bigwedge (A \cup A_{i_0}^{M \cup \{\alpha\}}) \wedge \phi \wedge \phi_i \Rightarrow h$$

It then follows immediately that

$$\mu F_{\mathcal{D}} \models \bigwedge A \wedge \phi \wedge \bigvee_{i=1}^m \left(\phi_i \wedge \left[\Gamma, A \cup A_{i_0}^{M \cup \{\alpha\}} \right]^{k_i} \wedge \bigwedge A_{i_0}^{M \cup \{\alpha\}} \right) \Rightarrow h$$

By Lemma 7, we get

$$\mu F_{\mathcal{D}} \models \llbracket \Gamma, A \rrbracket^{k'+1} \Rightarrow \bigvee_{i=1}^m \exists \tilde{x}_i. \left(\phi_i \wedge \left[\Gamma, A \cup A_{i_0}^{M \cup \{\alpha\}} \right]^{k'} \right),$$

where $k' = \max \{k_i\}_{i=1}^m$. Therefore, by Lemma 5, we obtain

$$\mu F_{\mathcal{D}} \models \llbracket \Gamma, A \rrbracket^{k'+1} \wedge \bigwedge A \wedge \phi \wedge \left(\bigvee_{i=1}^m (\phi_i \wedge \bigwedge A_i) \right) \Rightarrow h$$

From the facts $\mu F_{\mathcal{D}} \models P(\tilde{t}) \Leftrightarrow \bigvee_{i=1}^m \exists \tilde{x}_i. (\phi_i \wedge \bigwedge A_i)$ and $P_{\alpha}^M(\tilde{t}) \in A$, for $k = k' + 1$, we get

$$\mu F_{\mathcal{D}} \models \llbracket \Gamma, A \rrbracket^k \wedge \bigwedge A \wedge \phi \Rightarrow h.$$

• **Case APPLY \perp :** We have

- $(g, A', \phi', \perp) \in \Gamma$,
- $\text{dom}(\sigma) = \text{fus}(A')$,
- $\models \phi \Rightarrow \llbracket \sigma g \in A \rrbracket$,
- $\models \phi \Rightarrow \llbracket \sigma A' \subseteq A \rrbracket$,
- $\{\tilde{x}\} = \text{fus}(\phi') \setminus \text{dom}(\sigma)$, and
- $\mathcal{D}; \Gamma; A; \phi \wedge \forall \tilde{x}. \neg(\sigma \phi') \vdash h$.

By $(g, A', \phi', \perp) \in \Gamma$, $\models \phi \Rightarrow \llbracket \sigma g \in A \rrbracket$, and Lemmas 4 and 5, for some k_1 , we get

$$\models \llbracket \Gamma, A \rrbracket^{k_1} \wedge \bigwedge \sigma A' \wedge \phi \Rightarrow \forall \tilde{x}. \neg(\sigma \phi')$$

By $\models \phi \Rightarrow \llbracket \sigma A' \subseteq A \rrbracket$ and Lemma 3, we obtain

$$\models \llbracket \Gamma, A \rrbracket^{k_1} \wedge \bigwedge A \wedge \phi \Rightarrow \forall \tilde{x}. \neg(\sigma \phi')$$

By I.H., there is k_2 such that

$$\mu F_{\mathcal{D}} \models \llbracket \Gamma, A \rrbracket^{k_2} \wedge \bigwedge A \wedge \phi \wedge \forall \tilde{x}. \neg(\sigma \phi') \Rightarrow h$$

Therefore, for $k = \max(k_1, k_2)$, by Lemma 5, we obtain

$$\mu F_{\mathcal{D}} \models \llbracket \Gamma, A \rrbracket^k \wedge \bigwedge A \wedge \phi \Rightarrow h.$$

• **Case APPLY P :** We have

- $(g, A', \phi', P(\tilde{t})) \in \Gamma$,
- $\text{dom}(\sigma) = \text{fus}(A') \cup \text{fus}(\tilde{t})$,
- $\models \phi \Rightarrow \llbracket \sigma g \in A \rrbracket$,
- $\models \phi \Rightarrow \exists \tilde{x}. (\sigma \phi')$,

- $\models \phi \Rightarrow \llbracket \sigma A' \subseteq A \rrbracket$,
- $\{\tilde{x}\} = \text{fus}(\phi') \setminus \text{dom}(\sigma)$,
- $\mathcal{D}; \Gamma; A \cup \left\{ P_{\sigma}^{\emptyset}(\sigma \tilde{t}) \right\}; \phi \vdash h$

By $(g, A', \phi', P(\tilde{t})) \in \Gamma$, $\models \phi \Rightarrow \llbracket \sigma g \in A \rrbracket$, and Lemmas 4 and 5, for some k_1 , we get

$$\models \llbracket \Gamma, A \rrbracket^{k_1} \wedge \bigwedge \sigma A' \wedge \phi \wedge \exists \tilde{x}. (\sigma \phi') \Rightarrow P(\sigma \tilde{t})$$

Then, by $\models \phi \Rightarrow \llbracket \sigma A' \subseteq A \rrbracket$, Lemma 3, and $\models \phi \Rightarrow \exists \tilde{x}. (\sigma \phi')$, we get

$$\models \llbracket \Gamma, A \rrbracket^{k_1} \wedge \bigwedge A \wedge \phi \Rightarrow P(\sigma \tilde{t})$$

By I.H., there is k_2 such that

$$\mu F_{\mathcal{D}} \models \left[\Gamma, A \cup \left\{ P_{\sigma}^{\emptyset}(\sigma \tilde{t}) \right\} \right]^{k_2} \wedge \bigwedge (A \cup \left\{ P_{\sigma}^{\emptyset}(\sigma \tilde{t}) \right\}) \wedge \phi \Rightarrow h$$

By Lemma 6, we obtain

$$\mu F_{\mathcal{D}} \models \llbracket \Gamma, A \rrbracket^{k_2} \wedge P(\sigma \tilde{t}) \Rightarrow \left[\Gamma, A \cup \left\{ P_{\sigma}^{\emptyset}(\sigma \tilde{t}) \right\} \right]^{k_2}$$

It then follows that

$$\mu F_{\mathcal{D}} \models \llbracket \Gamma, A \rrbracket^{k_2} \wedge \bigwedge (A \cup \left\{ P_{\sigma}^{\emptyset}(\sigma \tilde{t}) \right\}) \wedge \phi \Rightarrow h$$

Therefore, for $k = \max(k_1, k_2)$, by Lemma 5, we get

$$\mu F_{\mathcal{D}} \models \llbracket \Gamma, A \rrbracket^k \wedge \bigwedge A \wedge \phi \Rightarrow h$$

• **Case FOLD:** We have

- $(P(\tilde{t}) \Leftarrow \phi' \wedge \bigwedge A') \in \mathcal{D}$,
- $\text{dom}(\sigma) = \text{fus}(A') \cup \text{fus}(\tilde{t})$,
- $\models \phi \Rightarrow \exists \tilde{x}. (\sigma \phi')$,
- $\models \phi \Rightarrow \llbracket \sigma A' \subseteq A \rrbracket$,
- $\{\tilde{x}\} = \text{fus}(\phi') \setminus \text{dom}(\sigma)$,
- $\mathcal{D}; \Gamma; A \cup \left\{ P_{\sigma}^{\emptyset}(\sigma \tilde{t}) \right\}; \phi \vdash h$

By $P(\tilde{t}) \Leftarrow \phi' \wedge \bigwedge A' \in \mathcal{D}$, $\text{dom}(\sigma) = \text{fus}(A') \cup \text{fus}(\tilde{t})$, and $\{\tilde{x}\} = \text{fus}(\phi') \setminus \text{dom}(\sigma)$, we obtain

$$\mu F_{\mathcal{D}} \models \bigwedge \sigma A' \wedge \exists \tilde{x}. (\sigma \phi') \Rightarrow P(\sigma \tilde{t})$$

By $\models \phi \Rightarrow \llbracket \sigma A' \subseteq A \rrbracket$, Lemma 3, and $\models \phi \Rightarrow \exists \tilde{x}. (\sigma \phi')$, we get

$$\mu F_{\mathcal{D}} \models \bigwedge A \wedge \phi \Rightarrow P(\sigma \tilde{t})$$

By I.H., there is k' such that

$$\mu F_{\mathcal{D}} \models \left[\Gamma, A \cup \left\{ P_{\sigma}^{\emptyset}(\sigma \tilde{t}) \right\} \right]^{k'} \wedge \bigwedge (A \cup \left\{ P_{\sigma}^{\emptyset}(\sigma \tilde{t}) \right\}) \wedge \phi \Rightarrow h$$

By Lemma 6, we obtain

$$\mu F_{\mathcal{D}} \models \llbracket \Gamma, A \rrbracket^{k'} \wedge P(\sigma \tilde{t}) \Rightarrow \left[\Gamma, A \cup \left\{ P_{\sigma}^{\emptyset}(\sigma \tilde{t}) \right\} \right]^{k'}$$

It then follows that

$$\mu F_{\mathcal{D}} \models \llbracket \Gamma, A \rrbracket^{k'} \wedge \bigwedge (A \cup \left\{ P_{\sigma}^{\emptyset}(\sigma \tilde{t}) \right\}) \wedge \phi \Rightarrow h$$

Therefore, for $k = k'$, we obtain

$$\mu F_{\mathcal{D}} \models \llbracket \Gamma, A \rrbracket^k \wedge \bigwedge A \wedge \phi \Rightarrow h$$

• **Case VALID \perp :** We have $h = \perp$ and $\models \phi \Rightarrow \perp$. Therefore,

$\mu F_{\mathcal{D}} \models \llbracket \Gamma, A \rrbracket^k \wedge \bigwedge A \wedge \phi \Rightarrow h$ holds for any k .

• **Case VALID P :** We have $h = P(\tilde{t})$ and $\models \phi \Rightarrow \llbracket P(\tilde{t}) \in A \rrbracket$.

By lemma 2, we get $\models \bigwedge A \wedge \phi \Rightarrow P(\tilde{t})$. It then follows that $\mu F_{\mathcal{D}} \models \llbracket \Gamma, A \rrbracket^k \wedge \bigwedge A \wedge \phi \Rightarrow P(\tilde{t})$ for any k .

□