Beyond the Elementary Representations of Program Invariants over Algebraic Data Types

Yurii Kostyukov
y.kostyukov@2015.spbu.ru
Saint Petersburg State University
JetBrains Research
Russia

Dmitry Mordvinov
dmitry.mordvinov@jetbrains.com
Saint Petersburg State University
JetBrains Research
Russia

Grigory Fedyukovich
grigory@cs.fsu.edu
Florida State University
USA

Abstract

First-order logic is a natural way of expressing properties of computation. It is traditionally used in various program logics for expressing the correctness properties and certificates. Although such representations are expressive for some theories, they fail to express many interesting properties of algebraic data types (ADTs). In this paper, we explore three different approaches to represent program invariants of ADT-manipulating programs: tree automata, and first-order formulas with or without size constraints. We compare the expressive power of these representations and prove the negative definability of both first-order representations using the pumping lemmas. We present an approach to automatically infer program invariants of ADT-manipulating programs by a reduction to a finite model finder. The implementation called RInGen has been evaluated against state-of-the-art invariant synthesizers and has been experimentally shown to be competitive. In particular, program invariants represented by automata are capable of expressing more complex properties of computation and their automatic construction is often less expensive.

CCS Concepts: • **Theory of computation → Invariants**; **Tree languages**; **Regular languages**; **Logic and verification**; **Automated reasoning**.

Keywords: invariants, first-order definability, tree automata, finite models, invariant representation, algebraic data types.

ACM Reference Format:
Yurii Kostyukov, Dmitry Mordvinov, and Grigory Fedyukovich. 2021. Beyond the Elementary Representations of Program Invariants over Algebraic Data Types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3453483.3454055

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLDI '21, June 20–25, 2021, Virtual, Canada
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8391-2/21/06…$15.00
https://doi.org/10.1145/3453483.3454055

1 Introduction

Specifying and proving properties of programs is traditionally achieved with the help of first-order logic (FOL). It is widely used in various techniques for verification, from Floyd-Hoare logic [24, 29] to constrained Horn clauses (CHC) [7] and refinement types [56]. The language of FOL allows to describe the desired properties precisely and make the verification technology accessible to the end user. Similarly, verification proofs, such as inductive invariants, procedure summaries, or ranking functions are produced and returned to the user also in FOL, thus facilitating the explainability of a program and its behaviors.

In this paper, we demonstrate theoretically and practically that the class of solutions traditionally considered by state-of-the-art FOL-based tools is not wide enough to fulfill the expectations from automated verification. Despite offering a high degree of expressiveness, decision procedures and constraint solvers are sensitive to particular FOL-fragments and have an emerging need for algorithmic improvement. Algebraic Data Types (ADT) enjoy a variety of decision procedures [4, 45, 48, 53] and Craig interpolation algorithms [30, 33], but still many practical tasks cannot be solved by state-of-the-art solvers for Satisfiability Modulo Theory (SMT) such as Z3, CVC4 [2], and Princess [51].

Furthermore, with the recent growth of the use of SMT solvers, it is often tempting to formulate verification conditions using the combination of different theories, e.g., as in [21]. Verification conditions could be expressed using the combination of ADT and the theory of Equality and Uninterpreted Functions (EUF). Although SMT solvers claim to support EUF, in reality the proof search process often hangs back attempting to conduct structural induction and discovering helper lemmas [57].

In this paper, we introduce a new *automata-based* class of representations of inductive invariants. The basic idea is to find a finite model of the verification condition and convert this model into a finite automaton. Instead of representing program states, finite models describe the states of the tree

automaton via a known correspondence between tree automata and finite models. The resulting representations of invariants are *regular* in a sense that they can "scan" the ADT term to the unbounded depth, which cannot be reached by the representations by first-order formulas (called *elementary* throughout the paper).

Our first contribution is the demonstration that regular invariants of ADT-manipulating programs could be constructed from finite models of the verification condition. Intuitively, the invariant generation problem can be reduced to the satisfiability problem of a formula constructed from the FOL-encoding of the program with pre- and post-conditions where uninterpreted symbols are used instead of ADT constructors. Although becoming an over-approximation of the original verification condition, it can be handled by existing finite model finders, such as Mace4 [44], Finder [52], Paradox [13], or CVC4 [50]. If satisfiable, the detected model is used to construct regular solutions of the original problem.

We have investigated the expressiveness of three different representations of program invariants: 1) state-of-the-art elementary representations, 2) first-order representations with size constraints, and 3) introduced in this paper regular representations. Our second contribution is the theoretical result on the negative definability of both first-order representations studies in the paper. Knowing about the undefinability lets us understand why SMT-based techniques like Z3 [35] or Eldarica [31] diverge on correct programs.

We have formulated two *pumping lemmas* for two first-order representations. The concept of pumping lemma arises in the universe of formal languages in connection with finite automata and context-free languages. Pumping lemmas state that for all languages in some class (e.g., regular languages, context-free languages), any big enough element can be "pumped", i.e., get an unbounded increase in some of its parts, and still generate to some elements of the language. Pumping lemmas are useful for proving negative definability: we suppose that a language belongs to some class, then apply a pumping lemma for that class and get some "pumped" element, which cannot be in the language, and by contradiction we deduce that the language does belong to that class.

We have implemented a tool called RInGen for automated inference of the regular invariants of ADT-manipulating programs and evaluated it against state-of-the-art inductive invariant generators that support ADT, namely Z3/Spacer [35], Eldarica [31], and VeriMAP-iddt [16]. It managed to find non-trivial invariants of various problems, including the inhabitance checking for simply typed lambda calculus (STLC).

2 Motivating Example

In this section, we demonstrate the phenomenon of *inexpressiveness* of a first-order language, in which program invariants can be represented. For example, the following program asserts that there are no two consecutive Peano numbers that are both even.

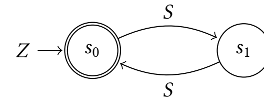
Example 1 (*Even*).

```
Nat ::= Z | S Nat
fun even(x : Nat) : bool =
    match x with
    | Z -> true
    | S Z -> false
    | S (S x') -> even(x')
assert ¬(∃x : Nat, even(x) ∧ even(S(x)))
```

This assertion holds, and the program is safe. One standard way to prove it is to discover a *safe inductive invariant*. The invariant could be represented by a first-order formula $even(x)$, satisfying the following logical constraints, which are called the *verification conditions* of the program in the form of constrained Horn clauses (CHC):

$$even(Z) \land \forall x.\big(even(x) \rightarrow even(S(S(x)))\big) \land$$
$$\forall x.\big(even(x) \land even(S(x)) \rightarrow \bot\big)$$

The first-order language of the Nat datatype is the language of successor arithmetic, i.e., the language, that allows for expressing only the first-order combination of arithmetic constraints of the form $x = c$ and $x = y + c$, where x and y are variables, and $c \in \mathbb{N}$. Thus, the successor arithmetic allows to define only finite and co-finite relations (i.e., relations with the finite complement) [19]. The only possible interpretation of *even* satisfying these CHCs is a relation $\{S^{2n}(Z) \mid n \geq 0\}$, which is not expressible in the first-order language of the Nat datatype. However, it could be represented by the automaton which moves to state s_0 for Z and flips the state from s_0 to s_1 and vice versa for S:



3 Preliminaries

Many-sorted logic. A many-sorted first-order signature with equality is a tuple $\Sigma = \langle \Sigma_S, \Sigma_F, \Sigma_P \rangle$, where Σ_S is a set of sorts, Σ_F is a set of function symbols, Σ_P is a set of predicate symbols, containing equality symbol $=_\sigma$ for each sort σ. Each function symbol $f \in \Sigma_F$ has an associated arity of the form $\sigma_1 \times \cdots \times \sigma_n \rightarrow \sigma$, where $\sigma_1, \ldots, \sigma_n, \sigma \in \Sigma_S$, and each predicate symbol $p \in \Sigma_P$ has an associated arity of the form $\sigma_1 \times \cdots \times \sigma_n$. Variables are associated with a sort as well. We use the usual definition of first-order terms with sort σ, ground terms, formulas, and sentences.

A many-sorted structure $\mathcal{M}$ for a signature Σ consists of non-empty domains $|\mathcal{M}|_\sigma$ for each sort $\sigma \in \Sigma_S$. For each function symbol f with arity $\sigma_1 \times \cdots \times \sigma_n \rightarrow \sigma$, it associates an interpretation $M(f) : |\mathcal{M}|_{\sigma_1} \times \cdots \times |\mathcal{M}|_{\sigma_n} \rightarrow |\mathcal{M}|_\sigma$, and for each predicate symbol p with arity $\sigma_1 \times \cdots \times \sigma_n$ it associated an interpretation $M(p) \subseteq |\mathcal{M}|_{\sigma_1} \times \cdots \times |\mathcal{M}|_{\sigma_n}$. For each ground term t with sort σ, we define an interpretation $\mathcal{M}[\![t]\!] \in |\mathcal{M}|_\sigma$ in a natural way. We call a structure finite if the domain of every sort is finite; otherwise, we call it infinite.

We assume the usual definition of a satisfaction of a sentence φ by $\mathcal{M}$, denoted $\mathcal{M} \models \varphi$. If φ is a formula, then we write $\varphi(x_1, \ldots, x_n)$ to emphasize that all free variables of φ are among $\{x_1, \ldots, x_n\}$. In this case, we denote the satisfiability $\mathcal{M} \models \varphi(a_1, \ldots, a_n)$ by $\mathcal{M}$ with free variables evaluated to elements $a_1, \ldots, a_n$ of the appropriate domains. The universal closure of a formula $\varphi(x_1, \ldots, x_n)$, denoted $\forall\varphi$, is the sentence $\forall x_1 \ldots \forall x_n.\varphi$. If φ has free variables, we define $\mathcal{M} \models \varphi$ to mean $\mathcal{M} \models \forall\varphi$.

A **Herbrand universe** for a sort σ is a set of ground terms with sort σ. If a Herbrand universe for a sort σ is infinite, we call σ an infinite sort. We say that $\mathcal{H}$ is a *Herbrand structure* $\mathcal{H}$ for a signature Σ if it associates the Herbrand universe $|\mathcal{H}|_\sigma$ to each sort σ of Σ as the domain and interprets every function symbol with itself, i.e., $\mathcal{H}(f)(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$ for all ground terms t_i with the appropriate sort. Thus, there is a family of Herbrand structures for one signature Σ with identical domains and interpretations of function symbols, but with various interpretations of predicate symbols. Every Herbrand structure $\mathcal{H}$ interprets each ground term t with itself, i.e., $\mathcal{H}[\![t]\!] = t$.

Assertion language. An algebraic data type (ADT) is a tuple $\langle C, \sigma\rangle$, where σ is a sort and C is a set of uninterpreted function symbols (called constructors), such that each $f \in C$ has a sort $\sigma_1 \times \cdots \times \sigma_n \to \sigma$ for some sorts $\sigma_1, \ldots, \sigma_n$.

In what follows, we fix a set of ADTs $\langle C_1, \sigma_1\rangle, \ldots, \langle C_n, \sigma_n\rangle$ with $\sigma_i \neq \sigma_j$ and $C_i \cap C_j = \varnothing$ for $i \neq j$. We define the signature[1] $\Sigma = \langle \Sigma_S, \Sigma_F, \Sigma_P\rangle$, where $\Sigma_S = \{\sigma_1, \ldots, \sigma_n\}$, $\Sigma_F = C_1 \cup \cdots \cup C_n$, and $\Sigma_P = \{=_{\sigma_1}, \ldots, =_{\sigma_n}\}$. For brevity, we omit the sorts from the equality symbols. We refer to the first-order language defined by Σ to as an *assertion language* $\mathcal{L}$.

Because Σ has no predicate symbols except the equality symbols (which have fixed interpretations within every structure), then there is a unique Herbrand structure $\mathcal{H}$ for Σ. We say that a sentence (a formula) φ in an assertion language is *satisfiable modulo theory* of ADTs, iff $\mathcal{H} \models \varphi$.

Constrained Horn Clauses. We refer to a finite set of predicate symbols $\mathcal{R} = \{P_1, \ldots, P_n\}$ with sorts from Σ to as *uninterpreted* symbols.

Definition 1. A constrained Horn clause (CHC) C is a $\Sigma \cup \mathcal{R}$-formula of the form:

$$\varphi \land R_1(\bar{t}_1) \land \ldots \land R_m(\bar{t}_m) \to H$$

where φ is a formula in the assertion language, called a *constraint*; $R_i \in \mathcal{R}$; $\bar{t}_i$ is a tuple of terms; and H, called a *head*, is either $\bot$, or an atomic formula $R(\bar{t})$ for some $R \in \mathcal{R}$.

If $H = \bot$, we say that C is a *query clause*, otherwise we call C a *definite clause*. The premise of the implication $\varphi \land R_1(\bar{t}_1) \land \ldots \land R_m(\bar{t}_m)$ is called a *body* of C.

A CHC system $\mathcal{S}$ is a finite set of CHCs.

[1] For simplicity, we omit the selectors and testers from the signature because they do not increase the expressiveness of the assertion language.

Satisfiability of CHCs. Let $\overline{X} = \langle X_1, \ldots, X_n\rangle$ be a tuple of relations, such that if P_i has sort $\sigma_1 \times \ldots \times \sigma_m$, then $X_i \subseteq |\mathcal{H}|_{\sigma_1} \times \ldots \times |\mathcal{H}|_{\sigma_m}$. To simplify the notation, we denote the expansion $\mathcal{H}\{P_1 \mapsto X_1, \ldots, P_n \mapsto X_n\}$ by $\langle \mathcal{H}, X_1, \ldots, X_n\rangle$, or simply by $\langle \mathcal{H}, \overline{X}\rangle$. We say that a system of CHCs $\mathcal{S}$ is *satisfiable modulo theory* of ADTs, if there exists a tuple of relations $\overline{X}$, such that $\langle \mathcal{H}, \overline{X}\rangle \models C$ for all $C \in \mathcal{S}$.

For example, the system of CHCs from Example 1 is satisfied by interpreting *even* with the relation

$$X = \{Z, S(S(Z)), S(S(S(S(Z)))), \ldots\} = \{S^{2n}(Z) \mid n \geq 0\}.$$

It is well known that CHCs provide a first-order match for a variety of program logics, including Floyd-Hoare logic for imperative programs and refinement types for higher-order functional programs. Thus, we assume that for every recursive program over ADTs there is a system of CHCs, such that the program is safe iff the system is satisfiable. In the rest of the paper, we use CHCs as a means of expressing verification conditions of programs.

Definability. A *representation class* is a function $\mathcal{C}$ mapping every tuple $\langle \sigma_1, \ldots, \sigma_n\rangle \in \Sigma_S^n$ for every $n \in \mathbb{N}$ to some class of languages $\mathcal{C}(\sigma_1, \ldots, \sigma_n) \subseteq 2^{|\mathcal{M}|_{\sigma_1} \times \cdots \times |\mathcal{M}|_{\sigma_n}}$. We say that a relation $X \subseteq |\mathcal{M}|_{\sigma_1} \times \ldots \times |\mathcal{M}|_{\sigma_n}$ is *definable* in a representation class $\mathcal{C}$ if $X \in \mathcal{C}(\sigma_1, \ldots, \sigma_n)$. We say that a Herbrand structure $\mathcal{H}$ is definable in $\mathcal{C}$ (or $\mathcal{C}$-definable) if for every predicate symbol $p \in \Sigma_P$ with arity $\sigma_1 \times \cdots \times \sigma_n$, interpretation $\mathcal{H}[\![p]\!]$ belongs to $\mathcal{C}(\sigma_1, \ldots, \sigma_n)$.

Finite Tree Automata. In order to define regular representations, we introduce *deterministic finite tree automata* (DFTA). Let $\Sigma = \langle \cdot, \Sigma_F, \cdot\rangle$ be a fixed many-sorted signature.

Definition 2 (cf. [14]). A *deterministic finite tree n-automaton* over Σ_F is a quadruple $(S, \Sigma_F, S_F, \Delta)$, where S is a finite set of states, $S_F \subseteq S^n$ is a set of final states, Δ is a transition relation with rules of the form:

$$f(s_1, \ldots, s_m) \to s,$$

where $f \in \Sigma_F$, $ar(f) = m$ and $s, s_1, \ldots, s_m \in S$, and there are no two rules in Δ with the same left-hand side.

Definition 3. A tuple of ground terms $\langle t_1, \ldots, t_n\rangle$ is *accepted* by n-automaton $A = (S, \Sigma_F, S_F, \Delta)$ iff $\langle A[t_1], \ldots, A[t_n]\rangle \in S_F$, where

$$A\big[f(t_1, \ldots, t_m)\big] \stackrel{\text{def}}{=} \begin{cases} s, & \text{if } \big(f(A[t_1], \ldots, A[t_m]) \to s\big) \in \Delta, \\ \bot, & \text{otherwise.} \end{cases}$$

Example 2. Let $\Sigma = \langle Prop, \{\land, \lor, \to, \top, \bot\}, \varnothing\rangle$ be a signature of propositional logic. The automaton A accepts true propositional formulas without variables:

$$A = (\{q_0, q_1\}, \Sigma_F, \{q_1\}, \Delta),$$

where the Δ is defined as follows:

$$\begin{array}{rr} \bot \mapsto q_0 & q_0 \lor q_0 \mapsto q_0 \\ \top \mapsto q_1 & * \lor * \mapsto q_1 \\ q_1 \land q_1 \mapsto q_1 & q_1 \to q_0 \mapsto q_0 \\ * \land * \mapsto q_0 & * \to * \mapsto q_1. \end{array}$$

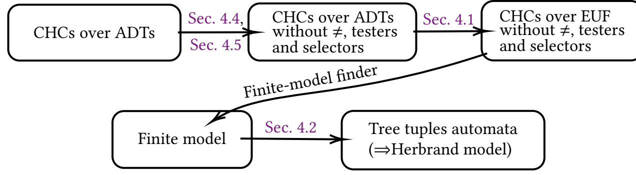


Figure 1. Obtaining regular model of a CHC system over ADTs.

Regular Herbrand Models. Let $\mathcal{H}$ be a Herbrand structure for a signature $\langle \cdot, \Sigma_F, \cdot \rangle$. We say that n-automaton A over Σ_F *represents* a relation $X \subseteq |\mathcal{H}|_{\sigma_1} \times \ldots \times |\mathcal{H}|_{\sigma_n}$ iff
$X = \{\langle a_1, \ldots, a_n \rangle \mid \langle a_1, \ldots, a_n \rangle \text{ is accepted by } A, a_i \in |\mathcal{H}|_{\sigma_i}\}$.
If there is a DFTA representing X, we call X *regular*. We denote the class of regular relations by Reg. A structure $\mathcal{H}$ is regular if it is Reg-definable.

4 Automated Inference of Regular Invariants

In this section, we show how to obtain regular models of CHCs over ADTs using a finite model finder, e.g., [13, 44, 50, 52]. Figure 1 gives the main steps of this process: Given a system of CHCs, we first rewrite it into a formula over uninterpreted function symbols by eliminating all disequalities, testers, and selectors from the clause bodies. Then we reduce the satisfiability modulo theory of ADTs to satisfiability modulo EUF and apply a finite model finder to build a finite model of the reduced verification conditions. Finally, using the correspondence between finite models and tree automata, we get the automaton representing the safe inductive invariant.

4.1 Translation to EUF

Recall that by definition, we call the system of CHCs over ADTs satisfiable if every clause is satisfied in some expansion of the Herbrand structure. The main insight is that this satisfiability problem can be reduced to checking the satisfiability of a formula over uninterpreted symbols in a usual first-order sense.

Informally, given a system of CHCs, we obtain another system by replacing all ADT constructors in all CHCs with uninterpreted function symbols. This allows the interpretations of constructors to violate the ADT axioms (distinctiveness, injectivity, exhaustiveness, etc.). The system with uninterpreted symbols is either satisfiable or unsatisfiable in the usual first-order sense. In the former case every clause is satisfied by some structure $\mathcal{M}$. It could be used to extract the interpretations of uninterpreted symbols in the Herbrand structure $\mathcal{H}$ which satisfy the original system over $\mathcal{H}$.

For instance, for the system of CHCs in Example 1, we check the satisfiability of the following formula:

$$\begin{aligned} &\forall x.(x = Z \to even(x)) \wedge \\ &\forall x, y.(x = S(S(y)) \wedge even(y) \to even(x)) \wedge \\ &\forall x, y.(even(x) \wedge even(y) \wedge y = S(x) \to \bot). \end{aligned}$$

The formula is satisfied by the following finite model $\mathcal{M}$:

$$|\mathcal{M}|_{Nat} = \{0, 1\} \qquad \mathcal{M}(Z) = 0$$
$$\mathcal{M}(even) = \{0\} \qquad \mathcal{M}(S)(x) = 1 - x.$$

4.2 Finite Models To Tree Tuples Automata

A procedure for constructing tree tuples automata (and, hence, regular models) from finite models follows immediately from the construction of an isomorphism between finite models and tree automata [41].

Given a finite structure $\mathcal{M}$, we build an automaton $\mathcal{A}_P = (|\mathcal{M}|, \Sigma_F, \mathcal{M}(P), \tau)$ for every predicate symbol $P \in \Sigma_P$. A shared set of transitions τ is defined as follows: for each $f \in \Sigma_F$ with arity $\sigma_1 \times \ldots \times \sigma_n \mapsto \sigma$, for each $x_i \in |\mathcal{M}|_{\sigma_i}$, $\tau(f(x_1, \ldots, x_n)) = M(f)(x_1, \ldots, x_n)$. For example, $\mathcal{A}_{even}$ is isomorphic to one shown in Example 1.

Theorem 1. *For the constructed automaton*

$$\mathcal{A}_P = (S, \Sigma_F, S_F, \tau),$$
$$L(\mathcal{A}_P) = \{ \langle t_1, \ldots, t_n \rangle \mid \langle \mathcal{M}[\![t_1]\!], \ldots, \mathcal{M}[\![t_n]\!] \rangle \in \mathcal{M}(P) \}.$$

In practice, this means that CHCs over ADTs could be automatically solved by *finite model finders*, such as Mace4 [44], Finder [52], Paradox [13], or CVC4 in a special mode [50]: if a *finite* model (in the usual first-order sense) is found, then there exists a *regular Herbrand* model of the CHC system. In Sec. 8, we evaluate our implementation that uses CVC4 against state-of-art CHC solvers.

4.3 Herbrand Models Without Equality

With the correspondence between finite models and tree automata in hand, it remains to show that the Herbrand model induced by the constructed tree automaton is a model of the original CHC system. In this subsection we show that it is straightforward when the system has no disequality constraints, otherwise some additional steps should be done.

Under assumption that the signature Σ of the assertion language does not have the equality symbol, there are no predicate symbols at all. Thus we may assume that every constraint in every CHC is $\top$. For instance, the above example could be rewritten to:

$$\begin{aligned} \top &\to even(Z) \\ even(x) &\to even(S(S(x))) \\ even(x) \wedge even(S(x)) &\to \bot. \end{aligned}$$

Lemma 2. *Suppose that a CHC system $\mathcal{S}$ over uninterpreted symbols $\mathcal{R} = \{P_1, \ldots, P_k\}$ with no constraints is satisfied by some first-order structure $\mathcal{M}$, i.e., $\mathcal{M} \models C$ for all $C \in \mathcal{S}$. Let*

$$X_i \stackrel{\text{def}}{=} \{\langle t_1, \ldots, t_n \rangle \mid \mathcal{M}[\![t_1]\!], \ldots, \mathcal{M}[\![t_n]\!] \in \mathcal{M}(P_i)\}.$$

Then $\langle \mathcal{H}, X_1, \ldots, X_k \rangle$ is the Herbrand model of $\mathcal{S}$.

Proof. As clause bodies have no constraints, each CHC is of the form $C \equiv R_1(\bar{t}_1) \wedge \ldots \wedge R_m(\bar{t}_m) \to H$. Then by definition

$$\langle \mathcal{H}, X_1, \ldots, X_k \rangle \models C \iff \mathcal{M} \models C,$$

so every clause in $\mathcal{S}$ is satisfied by $\langle \mathcal{H}, X_1, \ldots, X_k \rangle$. □

For Example 1, $X \overset{\text{def}}{=} \{t \mid \mathcal{M}[\![t]\!] = 0\} = \{S^{2n}(Z) \mid n \geq 0\}$ indeed satisfies the system.

4.4 Herbrand Models With Equality

In the presence of the equality symbol, which has the predefined semantics, a finite model finder searches for a model in a completely free domain, thus breaking the regular model. Consider the system consisting of the only CHC:

$$Z \neq S(Z) \rightarrow \bot.$$

This system is unsatisfiable because $\mathcal{H} \models Z \neq S(Z)$. But in a usual first-order sense, i.e., if we treat Z and S as uninterpreted functions, this CHC is satisfiable, e.g., as follows:

$$|\mathcal{M}|_{nat} = \{0\} \qquad \mathcal{M}(Z) = \mathcal{M}(S)(*) = 0$$

In general, every clause with a disequality constraint in the premise may be satisfied by falsifying its premise, e.g., by picking a sort with the cardinality one.

We propose the following way of attacking this problem. For every ADT $\langle C, \sigma \rangle$, we introduce a fresh uninterpreted symbol $diseq_\sigma$ and define $\mathcal{R}' \overset{\text{def}}{=} \mathcal{R} \cup \{diseq_\sigma \mid \sigma \in \Sigma_S\}$. Our process proceeds to constructing another system of CHCs $\mathcal{S}'$ over $\mathcal{R}'$. Without loss of generality, we may assume that the constraint of each clause $C \in \mathcal{S}$ is in the Negation Normal Form (NNF). Let C' be a clause with every literal of the form $\neg(t =_\sigma u)$ in the constraint (which we refer to as *disequality constraint*) substituted with the atomic formula $diseq_\sigma(t, u)$. For every clause $C \in \mathcal{S}$, we add C' into $\mathcal{S}'$. Finally, for every ADT $\langle C, \sigma \rangle$, we add the following rules for $diseq_\sigma$ to $\mathcal{S}'$:

- for all distinct c, c' of sort σ:
$$\top \rightarrow diseq_\sigma(c(\overline{x}), c'(\overline{x}'))$$
- for all constructors c of sort σ, x and y of sort σ', and i:
$$diseq_{\sigma'}(x, y) \rightarrow diseq_\sigma(c(\ldots, \underbrace{x}_{i\text{-th position}}, \ldots), c(\ldots, \underbrace{y}_{i\text{-th position}}, \ldots))$$

It is well-known that universal CHCs admit the least model, which is the denotational semantics of the program modeled by the CHCs, i.e., the least fixed point of the step operator. Thus, the following fact is trivial.

Lemma 3. *The rules of $diseq_\sigma$ have the least model over $\mathcal{H}$, which interprets $diseq_\sigma$ by a relation $\mathcal{D}_\sigma$, defined as $\mathcal{D}_\sigma \overset{\text{def}}{=} \left\{(x, y) \in |\mathcal{H}|^2_\sigma \mid x \neq y\right\}$ for each sort σ in Σ_S.*

As a corollary of this lemma, we state the following fact.

Lemma 4. *For a CHC system $\mathcal{S}$, let $\mathcal{S}'$ be a system with disequality constraints. Then, if $\langle \mathcal{H}, X_1, \ldots, X_k, Y_1, \ldots, Y_n \rangle \models \mathcal{S}'$, then $\left\langle \mathcal{H}, X_1, \ldots, X_k, \mathcal{D}_{\sigma_1}, \ldots, \mathcal{D}_{\sigma_n} \right\rangle \models \mathcal{S}'$ (here Y_i and $\mathcal{D}_{\sigma_i}$ interpret the $diseq_{\sigma_i}$ predicate symbol).*

Example 3. For $\mathcal{S} = \{Z \neq S(Z) \rightarrow \bot\}$, we get the following system of CHCs:

$$\begin{aligned}
\top &\rightarrow diseq_{Nat}(Z, S(x)) \\
\top &\rightarrow diseq_{Nat}(S(x), Z) \\
diseq_{Nat}(x, y) &\rightarrow diseq_{Nat}(S(x), S(y)) \\
diseq_{Nat}(Z, S(Z)) &\rightarrow \bot.
\end{aligned}$$

Recall that $\mathcal{S}$ is satisfiable in a usual first-order sense, but unsatisfiable in $\mathcal{H}$. But $\mathcal{S}'$ is unsatisfiable in a first-order sense since the query clause is derivable from the first rule, which solves our problem. In our workflow, we *search for finite models of $\mathcal{S}'$ instead of $\mathcal{S}$*, and then act as in the equality-free case. Finally, we end up with the following theorem:

Theorem 5. *Let $\mathcal{S}$ be a CHC system and $\mathcal{S}'$ be a CHC system with the disequality constraints. If there is a finite model of $\mathcal{S}'$ over EUF, then there is a regular Herbrand model of $\mathcal{S}$.*

Proof. We can rewrite CHCs into DNF, split them into different clauses and eliminate all equality atoms by the unification and substitution. Each clause $C \in \mathcal{S}$ then has the form:

$C \equiv u_1 \neq t_1 \wedge \ldots \wedge u_k \neq t_k \wedge R_1(\overline{x_1}) \wedge \ldots \wedge R_m(\overline{x_m}) \rightarrow H.$

In $\mathcal{S}'$, this clause becomes $C' \equiv$

$diseq(u_1, t_1) \wedge \ldots \wedge diseq(u_k, t_k) \wedge R_1(\overline{x_1}) \wedge \ldots \wedge R_m(\overline{x_m}) \rightarrow H.$

So, each clause in $\mathcal{S}'$ has no constraint (rules of *diseq* have no constraints as well), and by Lemma 2 there is a model $\langle \mathcal{H}, X_1, \ldots, X_k, U_1, \ldots, U_n \rangle$ of every $C' \in \mathcal{S}'$. Then, by Lemma 4 we have $\left\langle \mathcal{H}, X_1, \ldots, X_k, \mathcal{D}_{\sigma_1}, \ldots, \mathcal{D}_{\sigma_n} \right\rangle \models C'$. But

$\left\langle \mathcal{H}, X_1, \ldots, X_k, \mathcal{D}_{\sigma_1}, \ldots, \mathcal{D}_{\sigma_n} \right\rangle [\![C']\!] = \langle \mathcal{H}, X_1, \ldots, X_k \rangle [\![C]\!],$

thus giving us $\langle \mathcal{H}, X_1, \ldots, X_k \rangle \models C$ for every $C \in \mathcal{S}$. □

On finite model existence for CHCs with the disequality constraints. There is an interesting observation about finite models and disequality constraints. It can be (straightforwardly) shown that if ADT of sort σ has infinitely many terms, then the CHC

$$diseq_\sigma(x, x) \rightarrow \bot$$

is satisfied only by infinite structure, i.e., if we force the interpretations of *diseq* to omit the pairs of equal terms, then such system *has no finite models*. For comparison, if we force *diseq* to be false in just one tuple, the finite model may exist. For example, the query clause Q over the Nat datatype with

$$Q \equiv diseq_\sigma(Z, Z) \rightarrow \bot$$

is satisfiable in a finite model

$$|\mathcal{M}|_{Nat} = \{0, 1\}, \mathcal{M}(Z) = 0, \mathcal{M}(S)(*) = 1,$$
$$\mathcal{M}(diseq_{Nat}) = \{(0, 1), (1, 0), (1, 1)\}.$$

Intuitively, if for proving the satisfiability of CHCs we need to assume the disequality of a large number of ground terms, a finite model is less likely to exist. In practice, this means that tests containing disequality constraints are less likely to be satisfiable in some finite models. This is confirmed by our experimental evaluation in Sec. 8.

4.5 Removing Testers and Selectors

In practice, one can get verification conditions not only with constructors of algebraic datatypes, but also with testers and selectors. Unfortunately, a finite-model finder interprets its input as an EUF formula, and thus works in a completely free domain that breaks ADT axioms for testers and selectors. Similarly to the disequality case, we can deal with this problem by preprocessing a set of clauses and replacing testers and selectors with new clauses, e.g., for Lisp-style lists:

$$x = cons(r, y) \to car(x, r)$$
$$x = cons(y, r) \to cdr(x, r)$$
$$x = cons(y, z) \to cons?(x).$$

Thus, a clause

$$\neg(car(x) = cdr(y)) \to P(x, y)$$

can be rewritten as

$$car(x, a) \land cdr(y, b) \land \neg(a = b) \to P(x, y).$$

5 Case Study

In this section, we show a verification problem that was solved during experiments with our implementation (see Table 8). This case study demonstrates the expressiveness of regular representations. We also believe that this case may be interesting for readers interested in type theory.

Consider the following program sketch:

```
Var ::= ...
Type ::= arrow(Type, Type)
     | ... <primitive types> ...
Expr ::= var(Var) | abs(Var, Expr)
     | app(Expr, Expr)
Env ::= empty | cons(Var, Type, Env)

fun typeCheck(Γ: Env, e: Expr, t: Type): bool =
    match Γ, e, t with
    | cons(v, t, _), var(v), t -> true
    | cons(_, _, Γ'), var(_), _ ->
        typeCheck(Γ', e, t)
    | _, abs(v, e'), arrow(t, u) ->
        typeCheck(cons(v, t, Γ), e', u)
    | _, app(e1, e2), _ ->
        ∃u : Type, typeCheck(Γ, e2, u) ∧
                   typeCheck(Γ, e1, arrow(u, t))
    | _ -> false

assert ¬(∃e : Expr, ∀a, b : Type,
   typeCheck(empty, e, arrow(arrow(a, b), a)))
```

This program checks that there is no closed simply typed lambda calculus (STLC) term inhabiting the type $(a \to b) \to a$. A quantifier alternation is necessary here to show that there is no term with the *most general*, *principal* [28, def. 3A3] type $(a \to b) \to a$. It is well-known that this type is uninhabited, so this program is safe.

We wish to infer an invariant of `typeCheck` proving the validity of the assertion. Using the weakest liberal precondition calculus [18] we may obtain the verification conditions VC of this program, presented in the Figure 2.

VC is satisfiable modulo theory of algebraic data types `Var`, `Type`, `Expr` and `Env`, if and only if the program is safe. Moreover, the interpretations of *typeCheck* satisfying VC are the inductive invariants of the source program.

The strongest inductive invariant of the program is the least fixed point of a step operator, which is the set of all tuples (Γ, e, t), such that $\Gamma \vdash e : t$ in STLC typing rules. One needs a very expressive assertion language supporting type theory-specific reasoning to define this invariant. For example, this way is usually used in interactive theorem proving, when the STLC typing is defined in a sufficiently powerful type system of a proof assistant [11].

Instead, our goal is to verify this program automatically, using generic-purpose tools. So it is natural to look for coarser invariants. But does this program have weaker inductive invariants than $\{\langle\Gamma, e, t\rangle \mid \Gamma \vdash e : t\}$, still proving the validity of the assertion[2]? It turns out that the answer is yes, but it is difficult to compose this invariant. One surprisingly simple invariant $\mathcal{I}$ (see below) was discovered by our approach based on the finite model finding engine in CVC4 (see Sec. 4) completely automatically in less than a second.

Every STLC type can be viewed as a propositional formula, where atomic types correspond to atomic variables, and arrows correspond to implications. A *propositional interpretation* M for type t maps atomic variables of t to $\{0, 1\}$. We write $M \models t$ to denote that the propositional interpretation M satisfies the propositional formula corresponding to type t. We also say that type u is in $\Gamma \in Env$, if $\Gamma = cons(\ldots, cons(\cdot, u, \ldots))\ldots)$.

From the Curry-Howard correspondence, we know that the STLC type is inhabited if and only if the propositional formula defined by the type is a tautology of intuitionistic logic. But every intuitionistic tautology is the tautology of classical logic as well. So if type t is inhabited, then $M \models t$ for all propositional interpretations M. Thus, the following relation $\mathcal{I}$ over-approximates the strongest inductive invariant of the program:

$$\mathcal{I} \equiv \{\langle\Gamma, e, t\rangle \mid \text{for all } M, \text{either } M \models t, \text{or } M \not\models u \text{ for some type } u \text{ in } \Gamma\}.$$

Lastly, in our example $(a \to b) \to a$ is not a propositional tautology, and Γ is empty, so interpreting *typeCheck* with $\mathcal{I}$ satisfies the last clause of VC.

One could attempt to interpret *typeCheck* with relation

$$\mathcal{J} \equiv \{\langle\Gamma, e, t\rangle \mid t \text{ corresponds to a classical tautology}\},$$

but it fails because $\mathcal{J}$ is not inductive: e.g., it violates the first clause. Conversely, $\mathcal{I}$ satisfies all clauses. We can check that the first clause is satisfied by case splitting: if $M \models t$, then

[2] It should be noted that we did not find an answer to this question in the existing literature.

$$\forall \Gamma,\Gamma',e,t,v.\big(\Gamma = cons(v,t,\Gamma') \land e = var\ (v) \rightarrow typeCheck(\Gamma,e,t)\big)\land$$
$$\forall \Gamma,\Gamma',e,t,t',v,v'.\big(\Gamma = cons(v',t',\Gamma') \land e = var\ (v) \land (v \neq v' \lor t \neq t') \land typeCheck(\Gamma',e,t) \rightarrow typeCheck(\Gamma,e,t)\big)\land$$
$$\forall \Gamma,e,e',t,t',u,v.\big(e = abs(v,e') \land t = arrow(t',u) \land typeCheck(cons(v,t',\Gamma),e',u) \rightarrow typeCheck(\Gamma,e,t)\big)\land$$
$$\forall \Gamma,e,e_1,e_2,t,u.\big(e = app(e_1,e_2) \land typeCheck(\Gamma,e_2,u) \land typeCheck(\Gamma,e_1,arrow(u,t)) \rightarrow typeCheck(\Gamma,e,t)\big)\land$$
$$\forall e \exists a,b.\big(typeCheck(empty,e,arrow(arrow(a,b),a)) \rightarrow \bot\big)$$

Figure 2. Verification conditions VC of the $typeCheck$ program.

$\langle \Gamma, e, t\rangle \in \mathcal{I}$, otherwise $M \not\models t$, but t is in Γ by the premise of the clause, so again $\langle \Gamma, e, t\rangle \in \mathcal{I}$. Using the similar dichotomy, it is straightforward to check that $\mathcal{I}$ satisfies the remaining clauses.

Having the inductive invariant $\mathcal{I}$ in hand, it is still challenging to express it in the assertion language. With our new theoretical results for FOL-based languages we formally show in Appendix A that this invariant can not be defined in the first-order theory of algebraic data types `Var`, `Type`, `Expr` and `Env`, as well as any other safe inductive invariant of this program. For this reason, all state-of-art tools inferring the elementary inductive invariants fail for this program[3].

Instead, we could try to represent $\mathcal{I}$ by a tree automaton. First, there is an automaton, which determines if t is satisfied by a given interpretation M. This automaton has two states, 0 and 1. At the *arrow* constructor, it transits from a pair of states $(1, 0)$ to state 0, and to state 1 from the remaining pairs of states, modeling the logical implication. Starting from states corresponding to the interpretation of the leaves of t by M, the automaton stops at state 1 after scanning t iff $M \models t$.

Similarly, we can build an automaton which tests if there is a type u in Γ, such that $M \not\models u$. It contains two states, $\in$ and $\notin$. At the `empty` constructor, the automaton transits to the $\notin$ state. At the `cons` constructor, the automaton transits to the $\in$ state if it is already in the $\in$ state, or it is in the $\notin$ state, and the above automaton stops at 1 for the second argument of `cons`.

Formally, we have $\{\langle \Gamma, e, t\rangle \mid A \text{ accepts } \langle \Gamma, t\rangle\} \equiv \mathcal{I}$ for the tree automaton $A = \big(\{0, 1, \in, \notin, v, e\}, \Sigma_F, \{\langle \in, 0\rangle, \langle \notin, 1\rangle, \langle \in, 1\rangle\}, \Delta\big)$ with the following transition relation Δ:

$$\begin{aligned} Var_i &\mapsto v & arrow(1,0) &\mapsto 0 \\ PrimType_i &\mapsto 0 & arrow(*,*) &\mapsto 1 \\ var(v) &\mapsto e & empty &\mapsto \notin \\ abs(v,e) &\mapsto e & cons(v,1,\notin) &\mapsto \notin \\ app(e,e) &\mapsto e & cons(v,*,*) &\mapsto \in . \end{aligned}$$

In fact, if we replace the type $(a \rightarrow b) \rightarrow a$ in the program assertion by an arbitrary type t, which is not a tautology of classical logic, $\mathcal{I}$ still would prove the safety of the assertion. We have checked this experimentally. Note that $\mathcal{I}$ is simple enough to completely ignore the type-checked term e.

[3] Another reason is that the VC contains $\forall\exists$ quantifier alternation in the last clause. Although there are some attempts to design the decision procedure for CHCs with quantifier alternation (e.g., [5]), to the best of our knowledge all of them infer only elementary invariants and cannot handle this example.

One natural question regarding these invariants is what if we try an uninhabited type which corresponds to a classical tautology, but not to an intuitionistic one? One such example is the Pierce's law $t \equiv ((a \rightarrow b) \rightarrow a) \rightarrow a$. In this case $\mathcal{I}$ is too weak to prove that t is uninhabited. Our tool diverged on this input, which might mean that there is no regular inductive invariant, which over-approximates the denotational semantics of `typeCheck` and still proves the validity of the assertion. In the future, we will investigate it more thoroughly.

6 First-Order Invariant Representations

In this section, we compare the expressiveness of the regular representations with the first-order representations of invariants used in the state-of-the-art verification engines. We focus on two invariant representation languages: first-order formulas over ADTs (inferred, for example, by the fixed-point engine in Z3 [35]) and a richer language of first-order formulas over ADTs with size constraints used in Eldarica [31].

Although it is known that tree automata are equivalent to monadic second-order logic over trees [14], which is incomparable to FOL, it is still beneficial to study their relations in detail. Thus, our goal in this section is to come up with a vehicle for disproving relation definability in FOL. Studying the undefinable cases from the practical point of view lets us understand why provers like Z3 or Eldarica diverge for every safe program with undefinable invariants. In this section, we proceed by formulating and proving *pumping lemmas* for two first-order languages.

The concept of pumping lemma arises in the universe of formal languages in connection with finite automata and context-free languages. Pumping lemmas state that for all languages in some class (e.g., regular, context-free), any big enough element can be "pumped", i.e., get an unbounded increase in some of its parts and still stay in the language. Pumping lemmas are useful for proving undefinability: we assume that a language belongs to some class, apply a pumping lemma for that class and get some "pumped" element, which cannot be in the language. This by contradiction proves that the language does belong to that class.

6.1 Elementary Representations

We say that a relation $X \subseteq |\mathcal{M}|_{\sigma_1} \times \cdots \times |\mathcal{M}|_{\sigma_n}$ is *first-order definable*, or *elementary*, if there is a formula $\varphi(x_1, \ldots, x_n)$ in the assertion language, such that $(a_1, \ldots, a_n) \in X$ iff

$\mathcal{H} \models \varphi(a_1, \ldots, a_n)$. Let Elem be a representation class of elementary relations.

Example 4 (*IncDec*)**.** Consider the verification conditions of a simple program over Peano integers:

$$\begin{aligned} x = Z \land y = S(Z) &\rightarrow inc(x, y) \\ x = S(x') \land y = S(y') \land inc(x', y') &\rightarrow inc(x, y) \\ x = S(Z) \land y = Z &\rightarrow dec(x, y) \\ x = S(x') \land y = S(y') \land dec(x', y') &\rightarrow dec(x, y) \\ inc(x, y) \land dec(x, y) &\rightarrow \bot. \end{aligned}$$

The program has an obvious elementary invariant defined by $inc(x, y) \equiv (y = S(x))$, $dec(x, y) \equiv (x = S(y))$. This invariant is the strongest possible, i.e., it expresses the least fixed points of *inc* and *dec* correspondingly.

Interestingly, not every program has elementary-definable invariant[4], e.g., the undefinability of the strongest inductive invariant in Example 1 is shown by $E = \{Z, S(S(Z)), \ldots\}$. By extending it with some odd number $E \cup \{S^{2n+1}(Z)\} \subseteq E'$, we violate the query clause with $x = S^{2n}(Z)$ and $y = S^{2n+1}(Z)$. Thus we conclude that E is the only safe inductive invariant.

6.2 Pumping Lemma for Elem

In this subsection, we present our results on proving negative elementary definability: a pumping lemma for the first-order languages. We demonstrate it on the *Even* program (Example 1). Then we expand it to the first-order language with size constraints (SizeElem class) and use to prove the undefinability in SizeElem. To the best of our knowledge, this is the first undefinability result for SizeElem.

We begin with auxiliary definitions. The height of a ground term is defined inductively:

$$\mathcal{H}eight(c) \stackrel{\text{def}}{=} 1,$$
$$\mathcal{H}eight(c(t_1, \ldots, t_n)) \stackrel{\text{def}}{=} 1 + \max_{i=1}^{n} \left(\mathcal{H}eight(t_i)\right).$$

For each ADT $\langle \sigma, C \rangle$ and each constructor $f \in C$ having sort $\sigma_1 \times \cdots \times \sigma_n \rightarrow \sigma$ for some sorts $\sigma_1, \ldots, \sigma_n$, we define *selectors* $g_i \in S$ with sorts $\sigma \rightarrow \sigma_i$ for each $i \leq n$ with the standard semantics: $g_i(f(t_1, \ldots, t_n)) \stackrel{\text{def}}{=} t_i$.

A *path* is a (possibly empty) sequence of selectors: for $S_n : \sigma_n \rightarrow \sigma_{n-1}, \ldots, S_1 : \sigma_1 \rightarrow \sigma_0$, we define a path $s \stackrel{\text{def}}{=} S_1 \ldots S_n$. For terms t of sort σ_n, we define $s\,(t) \stackrel{\text{def}}{=} S_1(\ldots(S_n(t))\ldots)$. For ground terms g, we redefine $s\,(g)$ to be a computed subterm of g at s. We denote paths with small letters p, q, r, s.

We say that two paths p and q *overlap* if one of them is the suffix of the other. For pairwise non-overlapping paths $p_1, \ldots, p_n$, by $t[p_1 \leftarrow u_1, \ldots, p_n \leftarrow u_n]$ we denote the term obtained by simultaneous replacement of subterms $p_i\,(t)$ by u_i in t. For finite sequence of pairwise-distinct paths $P = (p_1, \ldots, p_n)$ and some terms $U = (u_1, \ldots, u_n)$, we abuse the notation and write $t[P \leftarrow U]$ meaning $t[p_1 \leftarrow u_1, \ldots, p_n \leftarrow u_n]$ and also $t[P \leftarrow t]$ for $t[p_1 \leftarrow t, \ldots, p_n \leftarrow t]$.

[4] In general, a program might have more than one inductive invariant. However, examples in this paper are designed to have unique inductive invariant.

Now we define a set of paths, which would be pumped.

Definition 4. For all sorts σ, we say that a term t is a *leaf term* of sort σ if it is either a base constructor, or $t = c(t_1, \ldots, t_n)$, where all t_i are leaf terms and t does not contain any proper subterms of sort σ. For a ground term g and a sort σ, we define $leaves_\sigma(g) \stackrel{\text{def}}{=} \{p \mid p\,(g) \text{ is a leaf term of sort } \sigma\}$.

Lemma 6 (Pumping Lemma for Elem)**.** *Let* **L** *be an elementary language of n-tuples. Then, there exists a constant $K > 0$ satisfying: for every n-tuples of ground terms $\langle g_1, \ldots, g_n \rangle \in \mathbf{L}$, for any i such that $\mathcal{H}eight(g_i) > K$, for all infinite sorts $\sigma \in \Sigma_S$ and for all paths p with a length greater than K, there exist finite sets of paths P_j such that $p \in P_i$, for all $p_1, p_2 \in \bigcup_j P_j$ it is true that $p_1(g) = p_2(g)$, and there is $N \geq 0$, such that for all t of sort σ with $\mathcal{H}eight(t) > N$ it holds that:*

$$\langle g_1[P_1 \leftarrow t], \ldots, g_i[P_i \leftarrow t], \ldots, g_n[P_n \leftarrow t] \rangle \in \mathbf{L}.$$

Proof. See the extended version [37, App. A.1]. □

Intuitively, Lemma 6 says that for big enough tuples of terms, we can take any of the deepest subterms, substitute them with *arbitrary* term t, and *still* obtain a tuple of terms in the language. The lemma formalizes our intuition that the first-order language of ADTs can only describe equalities and disequalities between subterms of a bounded depth: if one goes deep enough and replaces the leaf terms with the arbitrary terms, the initial and the resulting terms will be *indistinguishable* by the first-order language.

Now we demonstrate how to use the lemma to prove the undefinability results. Using pumping lemma, we prove that the invariant in the *Even* example is non-elementary.

Proposition 1. *Even* $\notin$ *Elem.*

Proof. Assume that there is the elementary safe inductive invariant **L**. Take a constant $K > 0$ from the pumping lemma. Let $g \equiv S^{2K}(Z) \in \mathbf{L}$, $\sigma = Nat$, $p = S^{2K}$. Now, $\bigcup_j leaves_\sigma(g_j) = leaves_\sigma(g) = \{p\}$, so $P = \{p\}$. Then, by pumping lemma there is $N \geq 0$, and we take $t \equiv S^{2N+1}(Z)$. By the lemma $g[P \leftarrow t] \equiv S^{2K}(S^{2N+1}(Z)) \in \mathbf{L}$. But then **L** violates the last clause of the verification conditions, thus **L** is not a safe inductive invariant, contradiction. □

As we have shown, pure elementary representations cannot express some program invariants. There are some attempts to increase the expressiveness of the first-order language by extending it with some additional symbols, still keeping the satisfiability of its formulas decidable.

6.3 Elementary Representations with Size Constraints

One natural idea to increase the expressiveness of the elementary representations is to extend the first-order language with the ability to specify *term sizes*. The inference of inductive invariants representable in that class is automated in the Eldarica CHC solver [31].

The language of SizeElem class can be derived from the language of Elem by adding into the signature a sort Int, Presburger arithmetic operations and function symbols $size_\sigma$ with arity $\sigma \rightarrow Int$, counting constructors in a term of sort σ. We omit σ from $size$ symbols for brevity.

The satisfiability of formulas with size constraints is checked in the $\mathcal{H}_{size}$ structure, obtained by conjoining the standard model of Presburger arithmetic to $\mathcal{H}$ and interpreting $size_\sigma$ function symbols straightforwardly. For example, given term

$$t \equiv cons(Z, cons(S(Z), nil))$$

with the sort $NatList := nil : NatList \mid cons : Nat \times NatList \rightarrow NatList$, we have $\mathcal{H}_{size}[\![size(t)]\!] = 6$.

We incorporate notations of Hojjat and Rümmer [30] and denote $\mathbb{T}_\sigma^k = \{t \text{ has sort } \sigma \mid size(t) = k\}$. For each ADT sort σ we define the set of term sizes $\mathbb{S}_\sigma = \{size(t) \mid t \in |\mathcal{H}|_\sigma\}$. A *linear set* is a set of the form $\{\mathbf{v} + \sum_{i=1}^n k_i \mathbf{v_i} \mid k_i \in \mathbb{N}_0\}$, where all $\mathbf{v}, \mathbf{v_i}$ are vectors over $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$.

Definition 5. An ADT sort σ is *expanding* iff for every natural number n there is a bound $b(\sigma, n) \geq 0$ such that for each $b' \geq b(\sigma, n)$, if $\mathbb{T}_\sigma^{b'} \neq \varnothing$, then $\left|\mathbb{T}_\sigma^{b'}\right| \geq n$. An ADT signature is called expanding if all its sorts are expanding.

Example 5 (*EvenLeft*)**.** Consider a tree datatype $Tree := leaf : Tree \mid node(Left : Tree, Right : Tree)$ and the program *EvenLeft* which checks if the leftmost branch of the tree has even number of nodes:

$$x = leaf \rightarrow EvenLeft(x)$$
$$x = node(node(x', y), z) \land EvenLeft(x') \rightarrow EvenLeft(x)$$
$$EvenLeft(x) \land EvenLeft(node(x, y)) \rightarrow \bot$$

Noteworthy, SizeElem admits quantifier elimination [58], so intuitively size constraints can only make restrictions on some subterms of finite depth, and they count all constructors at a time — we cannot specify to count only "the leftmost branch" constructors, as in example. More formally:

Lemma 7 (Pumping Lemma for SizeElem)**.** *Let the ADT signature be expanding and let* $\mathbf{L}$ *be an elementary language of* n*-tuples with size constraints. Then, there exists a constant* $K > 0$ *satisfying: for every* n*-tuple of ground terms* $\langle g_1, \ldots, g_n \rangle \in \mathbf{L}$*, for any* i*, such that* $\mathcal{H}eight(g_i) > K$*, for all infinite sorts* $\sigma \in \Sigma_S$*, and for all paths* $p \in leaves_\sigma(g_i)$ *with length greater than* K*, there exists an infinite linear set* $T \subseteq \mathbb{S}_\sigma$*, such that for all terms* t *of sort* σ *with sizes* $size(t) \in T$*, there exist sequences of paths* P_j*, with no path in them being a suffix of path* p*, and sequences of terms* U_j*, such that*

$$\langle g_1[P_1 \leftarrow U_1], \ldots, g_i[p \leftarrow t, P_i \leftarrow U_i], \ldots, g_n[P_n \leftarrow U_n] \rangle \in \mathbf{L}.$$

Proof. See the extended version [37, App. A.2]. ◻

Intuitively, having a SizeElem language, a big enough term g from it and deep enough path p, one can substitute $p\ (g)$ with almost arbitrary term t, limited only by size being in some linear, but infinite set T, and still obtain the term of the language. This means that for SizeElem languages there are unconstrained subterms, which are thus indistinguishable in this representation. Now we demonstrate the pumping lemma in action. To the best of our knowledge, this is the first negative definability result for SizeElem.

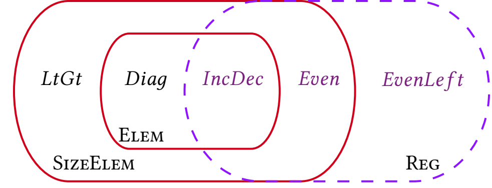


Figure 3. The comparison of the expressive power of three computable representations of inductive invariants. *LtGt* and *Diag* verification conditions can be found in the the extended version [37, App. C].

Proposition 2. *EvenLeft* $\notin$ *SizeElem.*

Proof. First, *Tree* sort is expanding. Let us assume *EvenLeft* is in SizeElem and it has an invariant $\mathbf{L}$. Take $K > 0$ from the lemma. Let $g \in \mathbf{L}$ be a full binary tree of height $2K$, $\sigma = Tree, p = Left^{2K}$. Take the infinite set of positive integers T from the lemma. We can take some $n \in T, n > 2$ and $t = node(leaf, t')$ for some t', such that $size(t) = n$. Now by the lemma, there exists a sequence of paths P and a sequence of terms U, and none of elements in P is a suffix of p. By the lemma, we must have $g[p \leftarrow t, P \leftarrow U] \in \mathbf{L}$, so the leftmost tree path must have an even length. However, at the leftmost path $p = Left^{2K}$ there is a term $node(leaf, t')$, so the leftmost tree path has length $2K - 1 + 2 = 2K + 1$, which is odd. Contradiction. ◻

In theory, size constraints add the expressive power to the assertion language. But in practice, it might happen that the automated inference of SizeElem invariants becomes harder because of checking the validity and generalization of size constraints. We evaluate Eldarica against other automated solvers to check that in Sec. 8.

7 Comparison of the Expressive Power

In this section, we compare the expressiveness of Elem, Reg and SizeElem classes. Our results are summarized in Figure 3. The regions in figure denote the representation classes: the small red one corresponds to the Elem class, the big red one stands for the SizeElem class, and the dotted purple region denotes the Reg class. The names in italic font denote programs. Each program inside a region denotes that the program has a safe inductive invariant definable in the corresponding class.

Due to the space limits, we skip the detailed proofs of the results in this picture. Instead, they can be found in the extended version [37, App. C]. Two proofs using our pumping lemmas have already been presented in Sec. 6.

Our results show that Reg and Elem classes, as well as Reg and SizeElem classes, are incomparable: there are programs with invariants definable in both classes and programs with invariants definable only in one of these classes. A quick

takeaway message is that first-order, logic-based representations and automata-based representations of relations in Herbrand universe are very different. While automata may specify precise enough properties of terms on unbounded depth, they fail to represent *relational properties* on terms (like disequality of terms or orderings over Peano numbers). On the other hand, while elementary representations are good for expressing the equality and disequality relations of terms, but can be precise only on a bounded depth.

Future Work. Regular relations can be further extended to other tree language classes (see, e.g., [9, 10, 20, 25, 32, 39]). Another intriguing prospect is to automate the inference of inductive invariants represented in first-order languages with regular language membership predicates [15]. This language is known to be decidable and closed under Boolean operations and subsumes both Reg and Elem classes.

8 Implementation and Experiments

We have evaluated our tool inferring regular invariants against state-of-the-art: Z3 inferring the Elem invariants and Eldarica using SizeElem representations on existing benchmarks. The amount of solved tasks correlates with definability: if a program has no invariants definable in a class, the corresponding solver diverges.

Implementation. We have implemented a regular invariant inference tool called RInGen[5] based on the preprocessing approach presented in Sec. 4. RInGen accepts input clauses in the SMTLIB2 [3] format and TIP extension with `define-fun-rec` construction [12]. It takes conditions with a property and checks if the property holds, returning SAT and the safe inductive invariant if it does or terminates with UNSAT if it does not. Thus RInGen can be run as a backend solver for functional program verifiers, such as MoCHi [34] and RCaml [55]. We run CVC4[6] as a backend multi sort finite-model finder to find regular models (Reg in our notation, see Example 3). Besides regular models, a finite model finding approach of CVC4 [50] based on quantifier instantiation provides us with sound unsatisfiability checking.

Benchmarks. We have empirically evaluated RInGen against state-of-the-art CHC solvers on "Tons of inductive problems" (*TIP*) benchmark set by Claessen et al. [12] and our own benchmarks inspired by the benchmark of De Angelis et al. [16]. The *TIP* benchmark set was preprocessed to be compatible with CHC solvers and the modified version[7] was contributed to the CHC-COMP solver competition[8].

We have modified the benchmark of De Angelis et al. [16] by replacing all non-ADT sorts with ADTs and adding CHC-definitions for non-ADT operations. The aggregated test set[9] consists of 60 CHC systems over binary trees, queues, lists, and Peano numbers. The test set was divided into two problem subsets *PositiveEq* and *Diseq*. *PositiveEq* is a set of CHC-systems with equality occurring only positively in clause bodies. *Diseq* set includes tests with occurrences of disequality constraints in clause bodies, substituted with `diseq` atoms, which is a sound transformation (see Sec. 4.4).

The *TIP* benchmark consists of 454 inductive ADT problems over lists, queues, regular expressions, and Peano integers originally generated from functional programs. From the original benchmark [12], we filtered out problems with only ADT sorts (the remaining problems use the combinations of ADTs with other theories), converted all of them into CHCs, replaced the disequalities with the `diseq` atoms as described in Sec. 4.4 and replaced all free sorts declared via `(declare-sort ... 0)` with the Nat datatype.

Competing tools. The evaluation was performed against Z3/Spacer [17] with the Spacer engine [36] and Eldarica [31] — state-of-the-art Horn-solvers which construct elementary models and support ADTs. Spacer works with elementary model representations (Elem in our notation, see Sec. 6.1). It incorporates standard decision, interpolation and quantifier elimination techniques for ADT [6]. Spacer is based on *property-directed reachability* (PDR), which alternates subtasks of counter-example finding and safe invariant construction by propagating reachability facts and pushing partial safety lemmas in a property-directed way.

Eldarica builds models with size constraints, which count the total number of constructor occurrences in them (named SizeElem in our notation, recall Sec. 6.3). It relies on the Princess SMT solver [51], which offers decision and interpolation procedures for ADT with size constraints by reduction to the combination of EUF and LIA [30].

As a baseline we include the CVC4 induction solver [49] into the comparison (denoted CVC4-Ind[10]), which leverages a number of techniques for inductive reasoning in SMT.

Finally, we provide a comparison against a VeriMAP extension called VeriMAP-iddt [16]. It handles the verification conditions over LIA and ADT and eliminates ADTs from the verification conditions completely by applying the fold/unfold techniques, so it is a clause transformer, which does not produce invariants over ADTs. Thus, it does not serve our main goal of comparing the expressivity of different invariant classes for ADT, however we include it as a baseline.

Experiments were performed on an Arch Linux machine Intel(R) Core(TM) i7-4710HQ CPU @ 2.50GHz 2.50GHz processor with 16GB RAM and a 300-second timeout.

Results. The results are summarized in Table 1.

On *PositiveEq* and *Diseq* benchmark sets, Spacer solved 7 problems and for the rest it ended with 8 UNKNOWN results

[5] *R*egular *In*variant *Gen*erator: https://github.com/Columpio/RInGen.

[6] Using `cvc4 --finite-model-find`.

[7] https://github.com/chc-comp/ringen-adt-benchmarks.

[8] https://chc-comp.github.io/

[9] https://gitlab.com/Columpio/adt-benchmarks/

[10] Using `cvc4 --quant-ind --quant-cf --conjecture-gen --conjecture-gen-per-round=3 --full-saturate-quant`

Table 1. Results of experiments on three ADT problem sets. Number in each cell stands for the amount of correct results within 300-seconds time limit. RInGen was used for *regular model* construction, Spacer was used for *elementary model* construction and Eldarica was used for building *elementary models with size constraints.* CVC4-Ind and VeriMAP-iddt are included for reference.

Invariant Representation			Reg	SizeElem	Elem	-	-
Problem Set	#	Answer	RInGen	Eldarica	Spacer	CVC4-Ind	VeriMAP-iddt
PositiveEq	35	SAT	27	1	4	0	0
Diseq	25	SAT	4	0	2	0	2
		UNSAT	1	1	1	1	1
TIP	454	SAT	30	46	26	0	16
		Unique SAT	13	25	7	0	0
		UNSAT	21	12	22	13	10
		Unique UNSAT	5	0	5	0	0
Total	514	SAT	61	47	32	0	18
		UNSAT	22	13	23	14	11

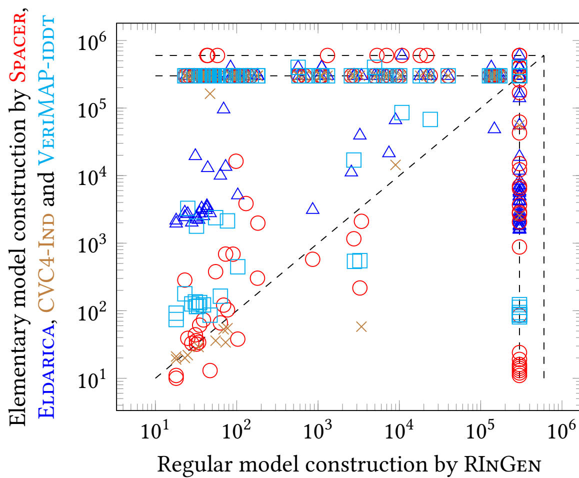


Figure 4. Comparison of engines performance. Each point in a plot represents a pair of the run times (sec × sec) of RInGen for Reg construction (x-axis) and a competitor for (Size)Elem construction (y-axis). Timeouts are placed on the inner dashed lines, crashes are on the outer dashed lines.

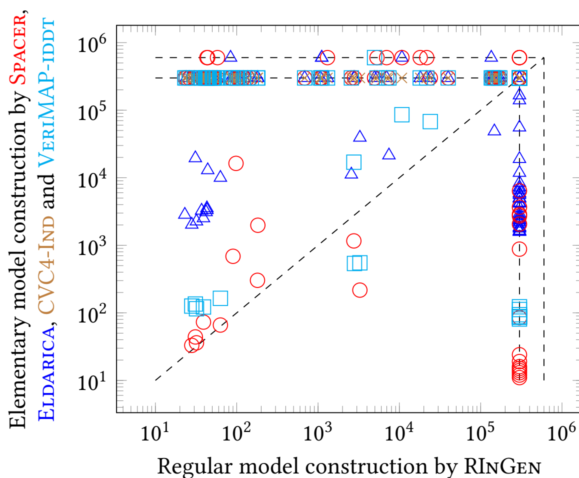


Figure 5. Comparison of engine performance with *only SAT results shown.* The testcase is included into this plot, if at least one of engines has discovered an invariant.

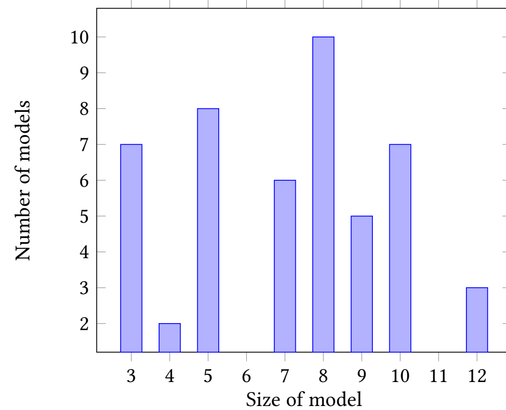


Figure 6. Sizes of finite models found in the evaluation. The size of model (x-axis) is calculated as the *sum* of all sorts cardinalities.

and 45 timeouts. Eldarica solved 2 problems (also solved by Spacer) with 58 timeouts. RInGen found 31 regular solutions, one counterexample and had 28 timeouts. Most of the solved problems are from *PositiveEq* test set, which does not contain equalities in the negative context. This confirm our hypothesis that such problems more likely have regular invariants, which is discussed in Sec. 4.4. Each problem solved by Spacer or Eldarica was solved by RInGen as well.

The *TIP* benchmark gave more diverse results. Firstly, all 12 problems claimed to be UNSAT by Eldarica were covered by RInGen as well, i.e., RInGen managed to find counterexamples more efficiently than Eldarica. RInGen and Spacer witnessed the unsatisfiability of 21 and 22 CHC-systems respectively. Most of these problems intersect, although some of them are unique for each solver.

Spacer exceeded the time limit 393 times. It terminated 13 times within time limit with the UNKNOWN result. RInGen exceeded the time limit 403 times, and Eldarica stopped after the time limit 368 times with 28 errors[11].

Finally, Eldarica proved safety with SAT result in 46 cases vs 30 cases of RInGen. They share 11 problems, on

[11] All with the same message: "`Cannot handle general quantifiers in predicates at the moment`".

which RInGen was two magnitudes faster. RInGen has 13 uniquely solved problems, all of them with some variant of evenness predicate on Peano numbers (e.g., "the length of list concatenation is even iff sum of list lengths is even"). This type of regularity is naturally handled by the finite-model finder (see the extended version [37, App. C, Prop. 8-9]). Eldarica has 25 uniquely solved problems, all of them with orderings ($<, \leq, >, \geq$) on Peano numbers. Note that this is exactly the case considered in the extended version [37, App. C, Prop. 11], differing SizeElem from the rest classes.

Timing plots in Figure 4 and Figure 5 show that not only RInGen inferred more invariants but it also was generally faster than other tools. On Figure 4, some unsafe benchmarks were handled faster by CVC4-Ind, VeriMAP-iddt and Spacer. This is possibly due to a more effective procedure of quantifier instantiation in CVC4-Ind and a more balanced tradeoff between the invariant inference and the counterexample search in the PDR core of Spacer (which is also called by the VeriMAP-iddt).

A diagram in Figure 6 shows the model sizes produced by the CVC4-f finite-model finder during the experiments.

Other experiments. We have tried 23 hand-written programs related to the type theory (recall Sec. 5), questioning the inhabitance of different STLC types, typability of STLC terms, and programs modeling different term-rewriting systems. All these benchmarks were intractable for all the solvers, except the finite model finder. For that reason, we omit the detailed statistics. We have also tried to run another finite model finders (for example, Mace4) as a backend, but they have shown worse results than CVC4.

Discussion. Clearly, finite model finding did much better on our own benchmarks inspired by the one of De Angelis et al. [16]. This is due to two reasons: the expressiveness of tree automata for representing the invariants and the efficiency of RInGen's backend CVC4-f engine. More importantly, Spacer and Eldarica diverged more often because of inexpressiveness of their FOL-based languages. Within the limits of their invariant representations, they perform smoothly.

On *TIP* benchmarks, Eldarica solved more testcases than RInGen, but the analysis of the testcases solved only by Eldarica has shown, that all such tests define the Peano ordering, easily handled by Eldarica by the reduction to LIA. On testcases solved by both engines, RInGen was faster in average. Still, the majority of interesting test cases in the *TIP* set obtained from proof assistants is currently beyond the reach of state-of-the-art engines under comparison, which motivates our future work.

To sum up, tree automata have been proven to be a promising technique in automated verification of ADT-manipulating programs. They allow to express complex properties of the recursive computation and can be efficiently inferred by the existing engines. In the future, however, a hybrid approach to infer invariants in parts by automata and in parts by FOL should exhibit the best performance.

9 Related Work

Language classes considered in this work have already been studied in the literature. Although these were separate works from different subfields of computer science.

Finite models and tree automata. A classic book on automated model building [8] gives a generous overview of finitely representable models and their features, decision procedures and closure properties. In the context of invariant inference it is important that checking the inductiveness of a candidate finite-model invariant is decidable and checking the existence of a finite model is not. The latter is semidecidable: we eventually terminate if a finite model exists, but do not terminate if the system has only infinite models.

Basic results for tree automata are accumulated in [14]. There is also an ongoing research on extensions of regular tree languages, which still enjoy nice decidability and closure properties [9, 10, 20, 25, 32, 39].

A number of tools like Mace4 [44], Finder [52], Paradox [13] and CVC4 [50] are used to find finite models of first-order formulas. Most of them implement a classic DPLL-like search with propagating assignments. CVC4, in addition, uses conflict analysis to accelerate the search. They were applied to various verification tasks [40] and even infinite models construction [46]. Yet we are unaware of applying finite model finders to inference of invariants of ADT-manipulating programs.

Recently, Haudebourg et al. [27] proposed a regular abstract interpretation framework for invariant generation for higher-order functional programs over ADTs. Authors derive a type system where each type is a regular language and use CEGAR to infer regular invariants. Their procedure is much more complex because they support high-order reasoning which is not the goal of this paper, comparing ADT-invariant representation. Targeting first-order functions over ADT only we obtain a more straightforward invariant inference procedure by using effective finite-model finders. Moreover, our work clarifies the gap between different invariant representations and their expressivity and aims not to advertise regular invariants themselves but to overcome mental inertia towards elementary invariant representations.

Herbrand model representations. There is a line of work studying different computable representations of Herbrand models [22, 23, 26, 54], which can be fruitful to study to find out new ADT invariant representations. Even though tree automata enjoy lots of effective properties, they have limited expressivity, so their extensions were widely studied in the automated model building field [8]. A survey on computational representations of Herbrand models, their properties, expressive power, correspondences and decision procedures can be found in [42, 43].

ADT solving. There is a plenty of proposed quantifier elimination algorithms and decision procedures for the first-order ADT fragment [4, 45, 47, 48, 53] and for an extension of ADT with constraints on term sizes [58]. Some works discuss the Craig interpolation of ADT constraints [30, 33]. Such techniques are being incorporated by various SMT solvers, like Z3 [17], CVC4 [2], and PRINCESS [51].

Some work on automated induction for ADT was proposed. Support for inductive proofs exists in deductive verifiers, such as DAFNY [38] and SMT solvers [49]. The technique in CVC4 is deeply integrated in the SMT level — it implements Skolemization with inductive strengthening and term enumeration to find adequate subgoals. De Angelis et al. [16] introduces a fold/unfold-based technique for eliminating ADTs from the CHC-system by transforming it to CHC-system over LIA and booleans. Recently, Yang et al. [57] applied a method based on Syntax-Guided Synthesis [1] to leverage induction by generating supporting lemmas based on failed proof subgoals and user-specified templates. On the whole, checking the inductiveness of FOL invariants is decidable as FOL is a decidable theory. Invariant inference is semidecidable in the sense that we can enumerate all candidate FOL-formulas but we do not terminate if the system has no FOL-representable invariant.

ADTs with size constraints. A brief survey of ADT theory with sized constraints is proposed by Zhang et al. [58]. In particular, quantifier elimination and decision procedures for this class are introduced. The automated inference of inductive invariants in SIZEELEM is implemented in ELDARICA Horn solver [31]. The ADT constraints are solved and generalized by the PRINCESS SMT solver [51], using the reduction to uninterpreted functions and linear integer arithmetic [30] with handling size constraints in spirit of Suter et al. [53].

10 Conclusion

We have compared the two branches of representing the invariants of programs manipulating ADTs: by the first-order formulas over ADTs, possibly extended with size constraints, or by tree automata. We have shown that these branches are incomparable, each with its own downsides and upsides.

We have demonstrated that tree automata are very promising for representing the invariants of computation over ADTs, as they allow to express properties of the unbound depth. On the downside, tree automata cannot express the relations between different variables.

Using the correspondence between finite models and tree automata, we were able to use the existing finite model finders for automated inference of regular inductive invariants. We have bypassed the problem of disequality constraints in the verification conditions and implemented a tool called RINGEN which automatically infers the regular invariants of ADT-manipulating programs. This tool is competitive with the state-of-art CHC solvers Z3/SPACER and ELDARICA. Using RINGEN, we have managed to detect interesting invariants of various inductive problems, including the non-trivial invariant of the inhabitance checking for STLC.

In the future, we mainly plan to investigate extensions of tree automata which subsume elementary invariants but still enjoy efficient decidability and closure properties.

A Analysis of the Case Study

In this appendix, we demonstrate the effectiveness of the "pumping" lemma machinery, introduced in Sec. 6.2, in proving negative definability for FOL-based languages. The demonstration is performed on a sophisticated $STLC$ type inhabitation example from Sec. 5.

For readability, let us denote $STLC$ terms and types in the classical manner: $\lambda x.\lambda y.x$, $a \to a$ and so on.

Proposition 3. *The CHC-system STLC from Figure 2 does not have a FOL-definable invariant.*

Proof. This fact can be proven by the application of the pumping lemma for the ELEM class. Let us prove that by contradiction. Suppose there is a FOL-definable invariant called **L**. With a constant K from the pumping lemma we can define

$$e \stackrel{\text{def}}{=} \lambda i_1 \ldots \lambda i_{K+1}.\lambda x.\lambda y.x.$$

Then for arbitrary ground term a of the ADT-sort $Type$ we can define

$$t_1 \stackrel{\text{def}}{=} \underbrace{(a \to a) \to \ldots \to (a \to a)}_{K+1 \text{ times}} \to (a \to a \to a).$$

It is easy to see from the typing rules that $\langle empty, e, t_1 \rangle \in \mathbf{L}$. Also $\mathcal{H}eight(t_1) > K$.

Let L and R also denote a domain and a codomain selectors of the typing arrow. Let us take $\sigma = Type$ and $p = R^{K+1}$, so $p(t_1) = (a \to a \to a)$. The lemma states that there should be finite sets of paths P_j such that $p \in P_i$ for some i, for all $p_1, p_2 \in \bigcup_j P_j$ it is true that $p_1(t_1) = p_2(t_1)$. Let us denote $P \stackrel{\text{def}}{=} \bigcup_j P_j$. As $p(t_1) = (a \to a \to a)$ and there is no such subterm elsewhere in t_1, $P = \{p\}$. For $N \geq 0$ from the lemma then we can define

$$t_2 \stackrel{\text{def}}{=} \underbrace{(a \to a) \to \ldots \to (a \to a)}_{N \text{ times}} \to a.$$

The lemma then states that $\langle empty, e, t_1[P \leftarrow t_2] \rangle \in \mathbf{L}$. Consider that

$$t_1[P \leftarrow t_2] = t_1[p \leftarrow t_2] = \underbrace{(a \to a) \to \ldots \to (a \to a)}_{K+N+1 \text{ times}} \to a.$$

By applying STLC application rule $K + N + 1$ times to e and $\lambda x.x$ term we obtain

$$\langle empty, e \underbrace{(\lambda x.x) \ldots (\lambda x.x)}_{K+N+1 \text{ times}}, a \rangle \in \mathbf{L}.$$

Thus, there exists a term which for any type a is typed with a in the empty context by **L**. In particular, this term is also typable by **L** with $(a \to b) \to a$ for arbitrary a and b, which contradicts the goal clause of the $STLC$ CHC system. ◻

References

[1] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *FMCAD*. IEEE, 1–17.

[2] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11) (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 171–177. Snowbird, Utah.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2017. *The SMT-LIB Standard: Version 2.6*. Technical Report. Department of Computer Science, The University of Iowa. Available at `www.SMT-LIB.org`.

[4] Clark Barrett, Igor Shikanian, and Cesare Tinelli. 2007. An abstract decision procedure for a theory of inductive data types. *Journal on Satisfiability, Boolean Modeling and Computation* 3 (2007), 21–46.

[5] Tewodros A Beyene, Corneliu Popeea, and Andrey Rybalchenko. 2013. Solving Existentially Quantified Horn Clauses. In *International Conference on Computer Aided Verification*. Springer, 869–882.

[6] Nikolaj Bjørner and Mikolás Janota. 2015. Playing with Quantified Satisfaction. *LPAR (short papers)* 35 (2015), 15–27.

[7] Nikolaj Bjørner, Ken McMillan, and Andrey Rybalchenko. 2013. On solving universally quantified horn clauses. In *International Static Analysis Symposium*. Springer, 105–125.

[8] Ricardo Caferra, Alexander Leitsch, and Nicolas Peltier. 2013. *Automated model building*. Vol. 31. Springer Science & Business Media.

[9] Jacques Chabin, Jing Chen, and Pierre Réty. 2006. *Synchronized-context free tree-tuple languages*. Technical Report. Citeseer.

[10] Jacques Chabin and Pierre Réty. 2007. Visibly pushdown languages and term rewriting. In *International Symposium on Frontiers of Combining Systems*. Springer, 252–266.

[11] Adam Chlipala. 2008. Parametric Higher-Order Abstract Syntax for Mechanized Semantics. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*. 143–156.

[12] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. 2015. TIP: tons of inductive problems. In *Conferences on Intelligent Computer Mathematics*. Springer, 333–337.

[13] Koen Claessen and Niklas Sörensson. 2003. New Techniques that Improve MACE-Style Finite Model Finding. In *Proceedings of the CADE-19 Workshop: Model Computation-Principles, Algorithms, Applications*. Citeseer, 11–27.

[14] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, C. Löding, S. Tison, and M. Tommasi. 2008. Tree Automata Techniques and Applications. Available on: https://jacquema.gitlabpages.inria.fr/files/tata.pdf. release November, 18th 2008.

[15] Hubert Comon and Catherine Delor. 1994. Equational formulas with membership constraints. *Information and Computation* 112, 2 (1994), 167–216.

[16] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. 2018. Solving Horn Clauses on Inductive Data Types Without Induction. *Theory and Practice of Logic Programming* 18, 3-4 (2018), 452–469.

[17] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[18] Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, Etats-Unis Informaticien, and Edsger Wybe Dijkstra. 1976. *A Discipline of Programming*. Vol. 1. Prentice-Hall Englewood Cliffs.

[19] Herbert Enderton and Herbert B Enderton. 2001. *A Mathematical Introduction to Logic*. Elsevier.

[20] Joost Engelfriet and Andreas Maletti. 2017. Multiple context-free tree grammars and multi-component tree adjoining grammars. In *International Symposium on Fundamentals of Computation Theory*. Springer, 217–229.

[21] Grigory Fedyukovich and Gidon Ernst. 2021. Bridging Arrays and ADTs in Recursive Proofs. In *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12652)*, Jan Friso Groote and Kim Guldstrand Larsen (Eds.). Springer, 24–42.

[22] Christian G Fermüller and Reinhard Pichler. 2005. Model representation via contexts and implicit generalizations. In *International Conference on Automated Deduction*. Springer, 409–423.

[23] Christian G Fermüller and Reinhard Pichler. 2007. Model representation over finite and infinite signatures. *Journal of Logic and Computation* 17, 3 (2007), 453–477.

[24] Robert W Floyd. 1967. Assigning Meanings to Programs. In *Proceedings of Symposium on Applied Mathematics*. Number 32.

[25] Valérie Gouranton, Pierre Réty, and Helmut Seidl. 2001. Synchronized tree languages revisited and new applications. In *International Conference on Foundations of Software Science and Computation Structures*. Springer, 214–229.

[26] Bernhard Gramlich and Reinhard Pichler. 2002. Algorithmic aspects of Herbrand models represented by ground atoms with ground equations. In *International Conference on Automated Deduction*. Springer, 241–259.

[27] Timothée Haudebourg, Thomas Genet, and Thomas Jensen. 2020. Regular language type inference with term rewriting. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 1–29.

[28] J. Roger Hindley. 1997. *Basic Simple Type Theory*. Cambridge University Press. https://doi.org/10.1017/CBO9780511608865

[29] Charles Antony Richard Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580.

[30] Hossein Hojjat and Philipp Rümmer. 2017. Deciding and interpolating algebraic data types by reduction. In *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE, 145–152.

[31] Hossein Hojjat and Philipp Rümmer. 2018. The ELDARICA horn solver. In *2018 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 158–164.

[32] Florent Jacquemard, Francis Klay, and Camille Vacher. 2009. Rigid tree automata. In *International Conference on Language and Automata Theory and Applications*. Springer, 446–457.

[33] Deepak Kapur, Rupak Majumdar, and Calogero G Zarba. 2006. Interpolation for Data Structures. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. 105–116.

[34] Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. 2011. Predicate abstraction and CEGAR for higher-order model checking. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 222–233.

[35] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2016. SMT-based model checking for recursive programs. *Formal Methods in System Design* 48, 3 (2016), 175–205.

[36] Anvesh Komuravelli, Arie Gurfinkel, Sagar Chaki, and Edmund M Clarke. 2013. Automatic abstraction in SMT-based unbounded software model checking. In *International Conference on Computer Aided Verification*. Springer, 846–862.

[37] Yurii Kostyukov, Dmitry Mordvinov, and Grigory Fedyukovich. 2021. Beyond the Elementary Representations of Program Invariants over Algebraic Data Types. arXiv:2104.04463 [cs.PL]

[38] K. Rustan M. Leino. 2012. Automating Induction with an SMT Solver. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation* (Philadelphia, PA) *(VMCAI'12)*. Springer-Verlag, Berlin, Heidelberg, 315–331.

[39] Sébastien Limet, Pierre Réty, and Helmut Seidl. 2001. Weakly regular relations and applications. In *International Conference on Rewriting*

Techniques and Applications. Springer, 185–200.

[40] Alexei Lisitsa. 2012. Finite models vs tree automata in safety verification. In *23rd International Conference on Rewriting Techniques and Applications (RTA'12)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[41] Robert Matzinger. 1997. Comparing computational representations of Herbrand models. In *Kurt Gödel Colloquium on Computational Logic and Proof Theory*. Springer, 203–218.

[42] Robert Matzinger. 1998. On computational representations of Herbrand models. *Uwe Egly and Hans Tompits, editors* 13 (1998), 86–95.

[43] Robert Matzinger. 2000. *Computational representations of models in first-order logic*. Ph.D. Dissertation. Technische Universität Wien, Austria.

[44] William McCune. 2003. Mace4 Reference Manual and Guide. *arXiv preprint cs/0310055* (2003).

[45] Derek C Oppen. 1980. Reasoning about recursively defined data structures. *Journal of the ACM (JACM)* 27, 3 (1980), 403–411.

[46] Nicolas Peltier. 2009. Constructing infinite models represented by tree automata. *Annals of Mathematics and Artificial Intelligence* 56, 1 (2009), 65–85.

[47] Tuan-Hung Pham, Andrew Gacek, and Michael W. Whalen. 2016. Reasoning About Algebraic Data Types with Abstractions. *J. Autom. Reasoning* 57, 4 (2016), 281–318.

[48] Andrew Reynolds and Jasmin Christian Blanchette. 2017. A decision procedure for (co) datatypes in SMT solvers. *Journal of Automated Reasoning* 58, 3 (2017), 341–362.

[49] Andrew Reynolds and Viktor Kuncak. 2015. Induction for SMT solvers. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 80–98.

[50] Andrew Reynolds, Cesare Tinelli, Amit Goel, and Sava Krstić. 2013. Finite model finding in SMT. In *International Conference on Computer Aided Verification*. Springer, 640–655.

[51] Philipp Rümmer. 2008. A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 274–289.

[52] John Slaney. 1994. FINDER: Finite Domain Enumerator System Description. In *International Conference on Automated Deduction*. Springer, 798–801.

[53] Philippe Suter, Mirco Dotta, and Viktor Kuncak. 2010. Decision Procedures for Algebraic Data Types with Abstractions. *Acm Sigplan Notices* 45, 1 (2010), 199–210.

[54] Andreas Teucke, Marco Voigt, and Christoph Weidenbach. 2019. On the expressivity and applicability of model representation formalisms. In *International Symposium on Frontiers of Combining Systems*. Springer, 22–39.

[55] Hiroshi Unno, Sho Torii, and Hiroki Sakamoto. 2017. Automating induction for solving horn clauses. In *International Conference on Computer Aided Verification*. Springer, 571–591.

[56] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. 2014. Refinement types for Haskell. In *ICFP*. ACM, 269–282.

[57] Weikun Yang, Grigory Fedyukovich, and Aarti Gupta. 2019. Lemma Synthesis for Automating Induction over Algebraic Data Types. In *International Conference on Principles and Practice of Constraint Programming*. Springer, 600–617.

[58] Ting Zhang, Henny B Sipma, and Zohar Manna. 2004. Decision procedures for recursive data structures with integer constraints. In *International Joint Conference on Automated Reasoning*. Springer, 152–167.