Solving Constrained Horn Clauses over ADTs by Finite Model Finding

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

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

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

Abstract

First-order logic is a natural way of expressing the properties of computation, traditionally used in various program logics for expressing the correctness properties and certificates. Subsequently, modern methods in the automated inference of program invariants progress towards the construction of first-order definable invariants. Although the first-order representations are very expressive for some theories, they fail to express many interesting properties of algebraic data types (ADTs).

Thus we propose to represent program invariants *regularly* with *tree automata*. We show how to automatically infer such regular invariants of ADT-manipulating programs using finite model finders. We have implemented our approach and evaluated it against the state-of-art engines for the invariant inference in first-order logic for ADT-manipulating programs. Our evaluation shows that automata-based representation of invariants is more practical than the one based on first-order logic since invariants are capable of expressing more complex properties of the computation and their automatic construction is less expensive.

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 [20, 24] to constrained Horn clauses (CHC) [6] and refinement types [50]. 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.

Algebraic Data Types (ADT) enjoy a variety of decision procedures [4, 39, 42, 47] and Craig interpolation algorithms [25, 28], 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 [45].

This **presentation-only** submission is based on the material concurrently submitted to another peer-reviewed conference.

With the recent growth of the use of SMT solvers, it is often tempting to formulate verification conditions using the combination of different theories. Specifically in the ADT case, 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 [51].

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. 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 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 [38], Finder [46], Paradox [12], or CVC4 [44]. If satisfiable, the detected model is used to construct regular solutions of the original problem.

We have implemented a tool called RegInv for automated inference of the regular invariants of ADT-manipulating programs and evaluated it against state-of-art inductive invariant generators, namely Z3/Spacer [30] and Eldarica [26] — the only CHC solvers supporting ADT, to the best of our knowledge. It managed to find non-trivial invariants of various problems, including the inhabitance checking for STLC.

2 Motivating Example

In this section we demonstrate one verification problem which is intractable for state-of-art solvers but is naturally handled by our approach. Basically, this case study demonstrates the expressiveness of regular representations in comparison to FOL-based ones. We believe that this case may

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

Figure 1. Verification conditions VC of the $typeCheck$ program.

be interesting from theoretical point of view for type theory experts.

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
    end

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 $(a \to b) \to a$. It is well-known that this type is uninhabited, so this program is safe.

Suppose, we wish to infer an inductive invariant of `typeCheck` proving the validity of the assertion. Using, for example, the weakest liberal precondition calculus [16], we may obtain the verification conditions VC of this program, presented in the Figure 1.

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 [10].

Instead, our goal is to verify this program automatically, using the 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[1]?

It turns out that the answer is yes, but it is not a simple task to compose this invariant. One surprisingly simple invariant $\mathcal{I}$ (see below) was discovered by our tool RegInv based on finite model finding engine in CVC4 (see Sec. 4) completely automatically in less than a second.

Every STLC type can be viewed as propositional formula, where type variables correspond to atomic variables, and arrows correspond to implications. Given type t, its *propositional interpretation* M is a map from 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 $\Gamma = cons(\dots, cons(\cdot, u, \dots))\dots)$.

Consider the following relation:

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

In the following, we explain the idea behind this invariant.

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 the type t is inhabited, then $M \models t$ for all propositional interpretations M. Thus, clearly, $\mathcal{I}$ over-approximates the strongest inductive invariant of the program. Also, 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: for instance, it violates the first clause. Conversely, $\mathcal{I}$ satisfies all clauses. The first clause is satisfied, which could be checked by case splitting: if $M \models t$, then $\langle \Gamma, e, t\rangle \in \mathcal{I}$, otherwise $M \not\models t$, but t is in Γ

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

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 rest clauses.

The invariant $\mathcal{I}$ could be represented 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, and after scanning the constructor $arrow(,)$ it transitions from a pair of states $(1, 0)$ to state 0, and to state 1 from the rest of pairs of states, modeling the logical implication. Starting from states corresponding to the interpretation of the leafs of t by M, the automaton stops in state 1 after scanning t iff $M \models t$.

Similarly, we can build the automaton which tests if there is a type u in Γ, such that $M \not\models u$. For this purpose, we need two states $\in$ and $\notin$. Scanning the `empty` constructor, the automaton transits to $\notin$ state. Scanning the `cons` constructor, the automaton transits to $\in$ state if it is already in $\in$ state, or it is in $\notin$ state, and the above automaton stops in 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 $(a \to b) \to a$ in the program assertion by the arbitrary type t, which is not a tautology of classical logic, $\mathcal{I}$ still would prove the safety of an assertion. We have checked this experimentally. Note that $\mathcal{I}$ is simple enough to completely ignore the type-checked term e.

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 $t \equiv ((a \to b) \to a) \to a$. In this case $\mathcal{I}$ is too weak to prove that t is uninhabited. Our tool diverged for 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. Although, that still should be investigated more thoroughly.

Thus, tree automata seem to be a balanced representation for ADT program invariants: they can express complex program properties and their inference can be efficiently automated. Regular invariants are formally defined in Sec. 3 and their automated inference with finite-model finders is described in Sec. 4. Our implementation and it's comparison against state-of-art on preexisted benchmarks is represented in Sec. 5.

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, among which there is a distinguished equality symbol $=_\sigma$ for each sort σ. Each function symbol $f \in \Sigma_F$ has associated with it an arity of the form $\sigma_1 \times \cdots \times \sigma_n \to \sigma$, where $\sigma_1, \ldots, \sigma_n, \sigma \in \Sigma_S$, and each predicate symbol $p \in \Sigma_P$ has associated with it an 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 $\sigma_1 \times \cdots \times \sigma_n \to \sigma$, it associates an interpretation $M(f) : |\mathcal{M}|_{\sigma_1} \times \cdots \times |\mathcal{M}|_{\sigma_n} \to |\mathcal{M}|_\sigma$, 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 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 the Herbrand universe for a sort σ is infinite, we call σ an infinite sort. We say that $\mathcal{H}$ is the *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[2] $\Sigma = \langle \Sigma_S, \Sigma_F, \Sigma_P\rangle$, 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

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

the sorts from the equality symbols. We refer to the first-order language defined by Σ to as an *assertion language* $\mathcal{L}$.

As Σ has no predicate symbols except the equality symbols (which have fixed interpretations within every structure), 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 $\langle C_1, \sigma_1\rangle, \ldots, \langle C_n, \sigma_n\rangle$, iff $\mathcal{H} \models \varphi$.

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

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

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

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 $\varphi \wedge R_1(\bar{t}_1) \wedge \ldots \wedge R_m(\bar{t}_m)$ is called a *body* of C.

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

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 $\langle \mathcal{H}, \overline{X}\rangle$.

Let $\mathcal{S}$ be a system of CHCs. We say that $\mathcal{S}$ 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 the 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 constrained Horn clauses provide a first-order match for lots of program logics, including Floyd-Hoare logic for imperative programs and refinement types for high-order functional programs. So, 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 article, we identify programs with their verification conditions expressed as systems of CHCs.

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 fixed many-sorted signature.

Definition 1 (cf. [13]). 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 2. 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

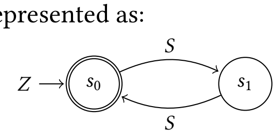
$$A\big[f(t_1, \ldots, t_m)\big] \stackrel{\text{def}}{=} \begin{cases} s, & \text{if } \big(f(A[t_1], \ldots, A[t_m]) \to s\big) \in \Delta, \\ \bot, & \text{otherwise.} \end{cases}$$

Example 1 (*Even*). For example, consider the following Peano integers datatype: $Nat := Z : Nat \mid S : Nat \to Nat$, and a CHC-system:

$$even(x) \leftarrow x = Z$$
$$even(x) \leftarrow x = S(S(y)) \wedge even(y)$$
$$\bot \leftarrow even(x) \wedge even(S(x))$$

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, the solution could be represented by the automaton $A = (\{s_0, s_1\}, \Sigma_F, \{s_0\}, \Delta)$ which moves to state s_0 for Z and flips the state from s_0 to s_1 and vice versa for S. The alphabet is simply $\Sigma_F = \{Z, S()\}$. The set of transition rules Δ can be represented as:



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

$X = \{\langle a_1, \ldots, a_n\rangle \mid \langle a_1, \ldots, a_n\rangle$ 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 demonstrate an approach to obtaining regular models of CHCs over ADTs using a finite model finder, e.g., [12, 38, 44, 46]. The main outline is shown in Figure 2.

The algorithm works in four steps. Given a system of constrained Horn clauses, we first rewrite it into a formula over uninterpreted function symbols by eliminating all disequalities from the clause bodies. Then we reduce the satisfiability modulo theory of ADTs to satisfiability modulo EUF and apply a finite model finder to construct 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 the replacement of all ADT constructors in all CHCs with uninterpreted function symbols. Thus we allow the interpretations of constructors to violate the ADT axioms (distinctiveness, injectivity, exhaustiveness, etc.). This system with uninterpreted symbols is either satisfiable or unsatisfiable in the usual first-order sense. If it is satisfiable, then every clause is satisfied by some structure $\mathcal{M}$. We could use this structure $\mathcal{M}$ to recover 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 the *even* example, we check the satisfiability of the following formula:

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

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

$$\begin{aligned}|\mathcal{M}|_{Nat} &= \{0, 1\} & \mathcal{M}(Z) &= 0\\ \mathcal{M}(even) &= \{0\} & \mathcal{M}(S)(x) &= 1 - x\end{aligned}$$

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 [35].

Given a finite structure $\mathcal{M}$, we construct an automaton $\mathcal{A}_P = \big(|\mathcal{M}|, \Sigma_F, \mathcal{M}(P), \tau\big)$ 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 $x_i \in |\mathcal{M}|_{\sigma_i}$, $\tau\big(f(x_1, \ldots, x_n)\big) = M(f)(x_1, \ldots, x_n)$.

Thus, for the *even* example we have $\mathcal{A}_{even}$ isomorphic to one introduced in Example 1.

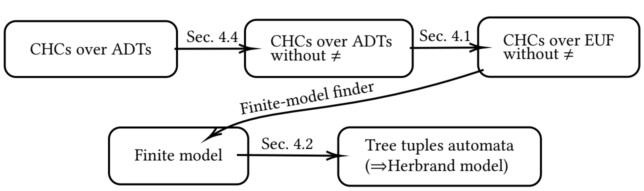


Figure 2. Obtaining regular model of a CHC system over ADTs.

Theorem 2. *For the constructed automaton* $\mathcal{A}_P = \big(S, \Sigma_F, S_F, \tau\big)$, $L(\mathcal{A}_P) = \{\langle t_1, \ldots, t_n\rangle \mid \big\langle \mathcal{M}[\![t_1]\!], \ldots, \mathcal{M}[\![t_n]\!]\big\rangle \in \mathcal{M}(P)\})$.

Proof. The proof is straightforward from the fact that $\mathcal{A}_P$ reflects checking the satisfiability in $\mathcal{M}$. □

In practice, this means that CHCs over ADTs could be automatically solved by *finite model finders*, such as Mace4 [38], Finder [46], Paradox [12] or CVC4 in a special mode [44]: 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. 5 we evaluate an implemented tool with the finite model finding engine in CVC4 as a backend 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, but otherwise some additional steps should be done.

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

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

Lemma 3. *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 $C \equiv R_1(\bar{t}_1) \land \ldots \land R_m(\bar{t}_m) \rightarrow H$. 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 the above example, we put $X \stackrel{\text{def}}{=} \{t \mid \mathcal{M}[\![t]\!] = 0\} = \{S^{2n}(Z) \mid n \geq 0\}$, indeed satisfying 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:

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

In general, every clause with a disequality constraint in the premise may be satisfied by falsifying its premise. It suffices to make the disequality false by picking a sort with the cardinality 1.

We propose the following way of attacking this problem. For every ADT (C, σ), we introduce a fresh uninterpreted symbol $diseq_\sigma$ and define $\mathcal{R}' \stackrel{\text{def}}{=} \mathcal{R} \cup \{diseq_\sigma \mid \sigma \in \Sigma_S\}$.

Below we present the process of 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 constraints*) 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 (C, σ), 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, all i, and x and y of sort σ':

$$diseq_{\sigma'}(x, y) \rightarrow diseq_\sigma(c(\dots, \underbrace{x}_{i\text{-th position}}, \dots), c(\dots, \underbrace{y}_{i\text{-th position}}, \dots))$$

Let $\mathcal{D}_\sigma \stackrel{\text{def}}{=} \{(x, y) \in |\mathcal{H}|^2_\sigma \mid x \neq y\}$ for each sort σ in Σ_S.

It is well-known that the 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 4. *The rules of $diseq_\sigma$ have the least model over $\mathcal{H}$, which interprets $diseq_\sigma$ by the relation $\mathcal{D}_\sigma$.*

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

Lemma 5. *For a CHC system $\mathcal{S}$, let $\mathcal{S}'$ be a system with the disequality constraints. Then, if $\langle \mathcal{H}, X_1, \dots, X_k, Y_1, \dots, Y_n \rangle \models \mathcal{S}'$, then $\left\langle \mathcal{H}, X_1, \dots, X_k, \mathcal{D}_{\sigma_1}, \dots, \mathcal{D}_{\sigma_n} \right\rangle \models \mathcal{S}'$ (here Y_i and $\mathcal{D}_{\sigma_i}$ interpret the $diseq_{\sigma_i}$ predicate symbol).*

Example 2. 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 6. *Let $\mathcal{S}$ be CHC system and $\mathcal{S}'$ be 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. Without loss of generality, we may assume that each clause $C \in \mathcal{S}$ is of the form (otherwise we rewrite the constraint into DNF, split it into different clauses and eliminate all the equality atoms by the unification and substitution):
$C \equiv y_1 \neq t_1 \wedge \dots \wedge y_k \neq t_k \wedge R_1(\overline{x_1}) \wedge \dots \wedge R_m(\overline{x_m}) \rightarrow H$.
In $\mathcal{S}'$, this clause becomes $C' \equiv$
$diseq(y_1, t_1) \wedge \dots \wedge diseq(y_k, t_k) \wedge R_1(\overline{x_1}) \wedge \dots \wedge R_m(\overline{x_m}) \rightarrow H$.

So, each clause in $\mathcal{S}'$ has no constraint (rules of $diseq$ have no constraints as well), and by Lemma 3 there is a model $\langle \mathcal{H}, X_1, \dots, X_k, Y_1, \dots, Y_n \rangle$ of every $C' \in \mathcal{S}'$. Then, by Lemma 5 we have $\left\langle \mathcal{H}, X_1, \dots, X_k, \mathcal{D}_{\sigma_1}, \dots, \mathcal{D}_{\sigma_n} \right\rangle \models C'$. But

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

thus giving us $\langle \mathcal{H}, X_1, \dots, X_k \rangle \models C$ 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 $diseq$ being 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, the chance of finite model existence is getting lower. In practice, this means that tests containing disequalities constraints have fewer chances to be satisfiable in some finite models. This is confirmed by our experimental evaluation (see Sec. 5).

5 Implementation and Experiments

We have evaluated our tool inferring regular invariants against state-of-art: Z3 and Eldarica on preexisted benchmarks.

Implementation. We have implemented a regular invariant inference tool called RegInv based on the preprocessing approach presented in Sec. 4 and an off-the-shelf finite-model finder [44]. RegInv accepts input clauses in SMTLIB2 [3] format and TIP extension with `define-fun-rec` construction [11]. It takes conditions with a property and checks if the property holds, returning safe inductive invariant if it does. Thus RegInv can be run as a backend solver for functional program verifiers, such as MoCHi [29] and RCaml [49].

RegInv can handle existentially-quantified Horn clauses. We run CVC4[3] as a backend multi sort finite-model finder to find regular models (see Sec.3).

Benchmarks. We empirically evaluate RegInv against state-of-art CHC solvers on benchmarks taken from works of Yang et al. [51], De Angelis et al. [14] and "Tons of inductive problems" (TIP) benchmark set by Claessen et al. [11].

We have modified the benchmarks of Yang et al. [51] and De Angelis et al. [14] by replacing all non-ADT sorts with ADTs (e.g., the *Int* sort in LIA with Peano integers using the Nat ADT) and adding CHC-definitions for non-ADT operations (for example, the addition was replaced by the addition of Peano numbers expressed as two CHCs). Thus, the aggregated testset[4] consists of 60 CHC systems over binary trees, queues, lists, and Peano numbers.

The test set was divided into two problem subsets, which we call *PositiveEq* and *Diseq*. *PositiveEq* is a set of CHC-systems with equality only occurring 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).

From *TIP* [11], we filtered out 377 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. Thus *TIP* benchmark consists of 377 inductive ADT problems over lists, queues, regular expressions, and Peano integers originally generated from functional programs.

Compared tools. The evaluation was performed against Z3/Spacer [15] with Spacer engine [31] and Eldarica [26] — state-of-art Horn-solvers which construct elementary models and support ADTs. Spacer works with elementary model representations. It incorporates standard decision, interpolation and quantifier elimination techniques for ADT [5]. Spacer is based on *property-directed reachability* (PDR) technique, which alternates counter-example finding and safe invariant construction subtasks 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. It relies on their own Princess SMT solver [45], which offers decision and interpolation procedures for ADT with size constraints by reduction to combination of AUF and LIA [25].

Finally, as a baseline we include the CVC4 induction solver [43] into the comparison (denoted CVC4-Ind[5]), which leverages a number of techniques for inductive reasoning in SMT.

[3]Using `cvc4 --finite-model-find`

[4]The link is omitted for the anonymity.

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

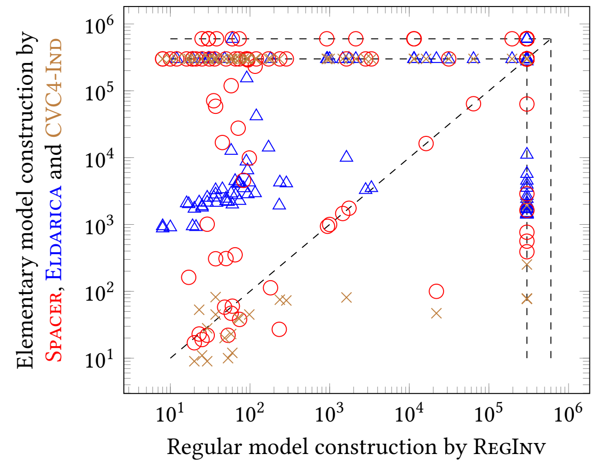


Figure 3. Comparison of engines performance. Each point in a plot represents a pair of the run times (sec × sec) of RegInv for Reg construction (x-axis) and a competitor for (Size)Elem construction (y-axis). Timeouts are placed on the inner dashed lines, crashes are on the outer dashed lines.

Problem Set	#	Answer	RegInv	Eldarica	Spacer	CVC4-Ind
PositiveEq	35	SAT	27	1	4	0
Diseq	25	SAT	4	0	2	0
		UNSAT	1	1	1	1
TIP	377	SAT	18	24	0	0
		UNSAT	36	40	31	22
Total	437	SAT	49	25	6	0
		UNSAT	37	41	32	23

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. RegInv was used for *regular model* construction, Spacer was used for *elementary model* construction and Eldarica was used for building *elementary models with size constraints.*

Despite the fact that we take first benchmarks from works of Yang et al. [51] and De Angelis et al. [14], we do not provide a comparison against tools from these papers. The main reason is that these tools are built on top of LIA solvers, and they do not produce invariants over ADTs. In particular, a tool from [14] handles the verification conditions over LIA and ADT and eliminates ADTs from the verification conditions completely. An approach of Yang et al. [51] is somewhat similar to CVC4-Ind and it handles LIA and EUF natively. So, these tools do not serve our main goal of comparing the expressivity of different invariant classes for ADT.

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 the *PositiveEq* and *Diseq* benchmark set, Spacer solved 7 problems and for the rest, it ended with 8 UNKNOWN results and 45 timeouts. Eldarica solved 2 problems (which were also solved by Spacer) with 58 timeouts. RegInv found 31 regular solutions, one counterexample and had 28 timeouts. Most of the solved problems are from *PositiveEq* test

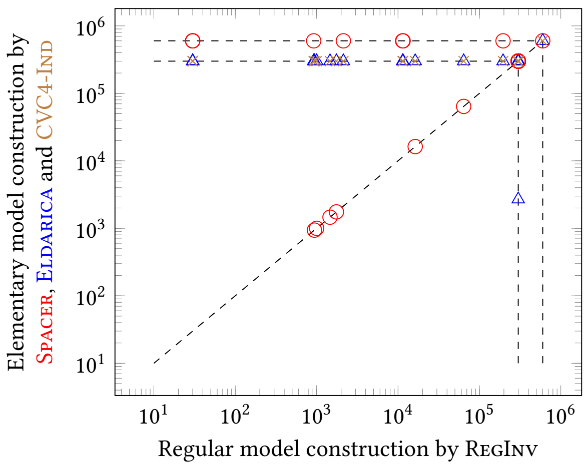


Figure 4. Comparison of engine performance with *only SAT results shown.* The testcase is included into this plot, if at least one of engines has discovered an invariant.

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 RegInv as well, i.e., if RegInv did not manage to prove the satisfiability within the time limit, none of the competitors succeeded as well.

The *TIP* benchmarks gave more diverse results. Firstly, all 36 problems claimed to be UNSAT by RegInv were covered by Eldarica as well, i.e., Eldarica managed to find counterexamples more efficiently that RegInv. Eldarica and Spacer witnessed the unsatisfiability of 40 and 31 CHC-systems respectively. Most of these problems intersect, although some of them are unique for each solver.

Spacer exceeded the memory limit of 16GB 64 times and the time limit 277 times. It terminated 5 times within time limit with the UNKNOWN result. RegInv exceeded the time limit 323 times with no other errors, and Eldarica stopped after the time limit 299 times with 14 errors[6].

Finally, Eldarica proved safety with SAT result in 24 cases vs 18 cases of RegInv. They share 12 problems; on shared problems RegInv was two magnitudes faster. RegInv has then 6 unique solved problems, all of them contain some variant of evenness predicate on Peano numbers (e.g., "the length of list concatenation is even iff sum of list lengths is even"), so this type of regularity is naturally handled by the finite-model finder. Eldarica has 12 unique (not solved by RegInv) problems, all of them with orderings ($<, \leq, >, \geq$) on Peano numbers, which is unsurprising as they are mapped to LIA through size constraints.

Timing plots in Figure 3 and Figure 4 show that not only RegInv inferred more invariants but it also was generally faster than other tools. On Figure 3, some unsafe benchmarks were handled faster by CVC4-Ind 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.

Other experiments. We have tried 23 hand-written programs related to the type theory (recall Sec. 2), 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 benchmarks from Yang et al. [51], De Angelis et al. [14] and our own experiments. This is due to two reasons: the expressiveness of tree automata for representing the invariants and the efficiency of RegInv'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 RegInv, 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 RegInv was faster in average. Still, lots of interesting test cases in the TIP set obtained from proof assistants are currently beyond the abilities of state-of-art engines under comparison.

From this evaluation we conclude that tree automata are very promising for automated verification of ADT-manipulating programs: they often allow to express complex properties of the recursive computation, and can be efficiently inferred by the existing engines.

6 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 Caferra et al. [7] gives a generous overview of finitely representable models and their features, like decision procedures and closure properties. Also, some results for tree automata and their extensions are accumulated in Comon et al. [13]. There is also an ongoing research on extensions of regular tree languages, which still enjoy nice decidability and closure properties [8, 9, 17, 21, 27, 33].

A number of tools, like Mace4 [38], Finder [46], Paradox [12] and CVC4 [44] are used to find finite models of

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

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 [34] and even infinite models construction [40]. Yet we are unaware of applying finite model finders to inference of invariants of ADT-manipulating programs.

Recently, Haudebourg et al. [23] proposed a regular abstract interpretation framework for invariant generation for high-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, this work makes clear 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 [18, 19, 22, 48], which can be fruitful to study to find out new ADT invariant representations. Even though tree automata enjoy lots of effective properties, they are limited in their expressive power, so a few of their extensions were widely studied by various researchers in the automated model building field [7]. A survey on computational representations of Herbrand models, their properties, expressive power, correspondences and decision procedures can be found in [36, 37].

ADT solving. There is a plenty of proposed quantifier elimination algorithms and decision procedures for the first-order ADT fragment [4, 39, 41, 42, 47] and for an extension of ADT with constraints on term sizes [52]. Some works discuss the Craig interpolation of ADT constraints [25, 28]. Such techniques are being incorporated by various SMT solvers, like Z3 [15], CVC4 [2] and Princess [45].

Some work on automated induction for ADT was proposed. Support for inductive proofs exists in deductive verifiers, such as Dafny [32] and SMT solvers [43]. The technique in CVC4 is deeply integrated in the SMT level — it implements Skolemization with inductive strengthening and term enumeration to find adequate subgoal. De Angelis et al. [14] introduces a technique for eliminating ADTs from the CHC-system by transforming it to CHC-system over integers and booleans. Recently, Yang et al. [51] 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.

7 Conclusion

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 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 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 this tool, we have managed to detect interesting invariants of various inductive problems, including the non-trivial invariant of the inhabitance checking for STLC.

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

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

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

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

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

[10] 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.

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

[12] 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.

[13] 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.
[14] 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.
[15] 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.
[16] 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.
[17] 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.
[18] Christian G Fermüller and Reinhard Pichler. 2005. Model representation via contexts and implicit generalizations. In *International Conference on Automated Deduction*. Springer, 409–423.
[19] Christian G Fermüller and Reinhard Pichler. 2007. Model representation over finite and infinite signatures. *Journal of Logic and Computation* 17, 3 (2007), 453–477.
[20] Robert W Floyd. 1967. Assigning Meanings to Programs. In *Proceedings of Symposium on Applied Mathematics*. Number 32.
[21] 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.
[22] 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.
[23] 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.
[24] Charles Antony Richard Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580.
[25] 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.
[26] Hossein Hojjat and Philipp Rümmer. 2018. The ELDARICA horn solver. In *2018 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 158–164.
[27] Florent Jacquemard, Francis Klay, and Camille Vacher. 2009. Rigid tree automata. In *International Conference on Language and Automata Theory and Applications*. Springer, 446–457.
[28] 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.
[29] 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.
[30] 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.
[31] 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.
[32] 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.
[33] Sébastien Limet, Pierre Réty, and Helmut Seidl. 2001. Weakly regular relations and applications. In *International Conference on Rewriting Techniques and Applications*. Springer, 185–200.
[34] 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.
[35] Robert Matzinger. 1997. Comparing computational representations of Herbrand models. In *Kurt Gödel Colloquium on Computational Logic and Proof Theory*. Springer, 203–218.
[36] Robert Matzinger. 1998. On computational representations of Herbrand models. *Uwe Egly and Hans Tompits, editors* 13 (1998), 86–95.
[37] Robert Matzinger. 2000. *Computational representations of models in first-order logic*. Ph.D. Dissertation. Technische Universität Wien, Austria.
[38] William McCune. 2003. Mace4 Reference Manual and Guide. *arXiv preprint cs/0310055* (2003).
[39] Derek C Oppen. 1980. Reasoning about recursively defined data structures. *Journal of the ACM (JACM)* 27, 3 (1980), 403–411.
[40] Nicolas Peltier. 2009. Constructing infinite models represented by tree automata. *Annals of Mathematics and Artificial Intelligence* 56, 1 (2009), 65–85.
[41] 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.
[42] 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.
[43] Andrew Reynolds and Viktor Kuncak. 2015. Induction for SMT solvers. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 80–98.
[44] 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.
[45] 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.
[46] John Slaney. 1994. FINDER: Finite Domain Enumerator System Description. In *International Conference on Automated Deduction*. Springer, 798–801.
[47] Philippe Suter, Mirco Dotta, and Viktor Kuncak. 2010. Decision Procedures for Algebraic Data Types with Abstractions. *Acm Sigplan Notices* 45, 1 (2010), 199–210.
[48] 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.
[49] Hiroshi Unno, Sho Torii, and Hiroki Sakamoto. 2017. Automating induction for solving horn clauses. In *International Conference on Computer Aided Verification*. Springer, 571–591.
[50] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. 2014. Refinement types for Haskell. In *ICFP*. ACM, 269–282.
[51] 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.
[52] 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.