Solving Constrained Horn Clauses over Algebraic Data Types

Lucas Zavalía, Lidiia Chernigovskaia, and Grigory Fedyukovich(✉)

Florida State University, Tallahassee, FL, USA
lrzavalia@fsu.edu, lidiya.chernigovskaya@gmail.com, grigory@cs.fsu.edu

Abstract. Safety verification problems are often reduced to solving the satisfiability of Constrained Horn Clauses (CHCs), a set of constraints in first-order logic involving uninterpreted predicates. Synthesis of interpretations for the predicates, also known as *inductive invariants synthesis*, is challenging in the presence of Algebraic Data Types (ADTs). Defined inductively, ADTs describe possibly unbounded chunks of data, thus they often require synthesizing recursive invariants. We present a novel approach to this problem based on *functional synthesis*: it attempts to extract recursive functions from constraints that capture the semantics of unbounded computation over the chunks of data encoded in CHCs. Recursive function calls are beneficial since they allow rewriting the constraints and introducing equalities that further can be simplified away. This largely simplifies the problem of generating invariants and lets them have simple interpretations that are recursion-free at the highest level and have function calls. We have implemented the approach in a new CHC solver called AdtChc. Our algorithm relies on an external automated theorem prover to conduct proofs by structural induction, as opposed to a black-box constrained solver. With two alternative solvers of choice, AdtInd and Vampire, the new toolset has been evaluated on a range of public benchmarks, and it exhibited its strengths against state-of-the-art CHC solvers on particular benchmarks that require recursive invariants.

1 Introduction

The trend in programming languages to organize data recursively originates from the fundamentals of logics, and it was first proposed as an alternative to pointers by Hoare [21] almost half a century ago. Since then, algebraic data types (ADTs) found their use as a modeling language in various software verification problems and enjoy tailored decision procedures [3,45–47,55]. With recursive functions over ADTs, verification conditions have a compact and elegant structure and can be handled by structural induction. However, induction-based methods often require adding helper lemmas that themselves require proofs [48,59]. Recent approaches to lemma synthesis are based on Satisfiability Modulo Theories (SMT) and suggest using proof-failure generalization and Syntax Guided

The second author is currently employed at JetBrains N.V., The Netherlands.

C. Dragoi et al. (Eds.): VMCAI 2023, LNCS 13881, pp. 341–365, 2023.
https://doi.org/10.1007/978-3-031-24950-1_16

Synthesis (SyGuS) [1]. It still needs improvements both in terms of scalability and expressiveness of the supported formulas.

Constrained Horn clauses (CHCs) over ADTs serve as a model for recursive computation and enable formulating safety verification tasks. CHCs make use of uninterpreted predicate symbols and a set of first-order logic implications that can use these predicates either in their left-hand side, right-hand side, or in both. Cyclic logic relations, formulated this way, correspond to loops and recursive functions. Interpretations to predicate symbols that satisfy all implications, can be treated as inductive invariants. In fact, the applicability of CHCs goes far beyond deductive verification conditions over traditionally defined recursive data structures. In software model checking [25], ADTs might encode strings, in synthesis problems [32] – the unrealizability, in relational verification [16] – simulation relations. That is, a CHC solver gradually becomes a *push-the-button technique* applicable in many domains, and thus it exempts the user from doing any specific preparation of the code and providing insights to the solver. A richer arsenal of low-level approaches that a CHC solver might employ is therefore required, e.g., new approaches to *functional synthesis* that are capable of extracting a function definition from a declarative specification.

Although there are many CHC solvers [2,5,8,9,13,19,28,29,34,38,40,41,44,51,57,61] available for various SMT theories (e.g., integer/real arithmetic, bitvectors, and/or arrays), only a few solvers, e.g., [11,24,52], can actually support ADTs. In fact, there is a big challenge while solving CHCs for ADTs. Because ADTs are defined inductively, all the functions that process them need to be recursive too. To capture the behavior of these recursive functions over ADTs, invariants often need to describe properties over all elements of these ADTs. Specifically, this often requires the invariant to be recursive itself, thus allowing one to express properties over potentially unbounded data structures. However, when validating such invariants, a (set of) universally quantified formula(s) over ADTs needs to be constructed, and an automated proof checker should conduct the validity proofs by structural induction.

Our contribution lies in the approach to generate recursive invariants over inductively defined data structures that capture the semantics of recursive functions precisely. In particular, our approach seeks to extract a functional representation from the CHC constraints and exploit an automated theorem prover to validate this functional representation with respect to the given safety property. Our solver called AdtChc builds on top of recent advances of automated theorem proving [36,59] that are capable of validating the interpretations constructed by the invariant synthesizer on the fly. Provers split a goal into a base case and an inductive step, prove each of them separately, generate and exploit inductive hypotheses. Whenever needed, provers can also generate a set of helper lemmas to be used for future subgoals.

Our secondary contribution in this paper is in the amendments to the AdtInd [59] prover that is the primary backend solver of AdtChc. We present two new features of AdtInd that help in its proving process: generation of helper lemmas from common subterms and filtering possibly invalid lemmas. These features are needed when a current subgoal requires an additional induction, which could be expensive. We thus synthesize a candidate lemma and attempt to prove it by induction, such that if it is successful then the lemma helps to prove the ultimate goal. However, during the synthesis, there are often many invalid lemma candidates. Our approach thus relies on a filtering procedure to remove some lemma candidates quickly.

AdtChc and AdtInd are built on top of the Z3 SMT solver [12]. In addition to ADTs, they support constraints in linear arithmetic and uninterpreted functions. We have evaluated AdtChc on a range of public benchmarks originated from the safety verification tasks written in functional programming languages. We have compared AdtChc to the top of CHC solvers presented in the CHC-COMP [50], namely Eldarica [24], HOICE [8], PCSat [52] and Racer [27] (implemented on top of GSpacer [37]). The experiments show that our tool is able to solve more benchmarks than competitors.

2 Preliminaries

A many-sorted first-order theory is defined as a tuple $\langle \mathcal{S}, \mathcal{F}, \mathcal{P} \rangle$, where $\mathcal{S}$ is a set of sorts, $\mathcal{F}$ is a set of function symbols, and $\mathcal{P}$ is a set of predicate symbols, including equality. A formula φ is called satisfiable if there exists a model where φ evaluates to *true*. If every model of φ is also a model of ψ, then we write $\varphi \implies \psi$. A formula φ is called *valid* if $true \implies \varphi$.

Definition 1 (ADT). An ADT is a tuple $\langle s, C \rangle$, where s is a sort and C is a set of uninterpreted functions (called *constructors*), such that each $c \in C$ has some type $A \to s$ for some A. If for some s, A is s-free, we say that c is a *base* constructor denoted bc_s (otherwise, an *inductive* constructor denoted ic_s).

In this paper, we assume that all ADTs are *well-defined* in the sense that for each a of sort s, if a is constructed using some $c_i \in C$, i.e., $\exists b \,.\, a = c_i(b)$ is true, then for all other constructors $c_j \in C \setminus \{c_i\}$, $\forall b \,.\, a \neq c_j(b)$ is true. Well-definedness allows for pattern matching, which is the key vehicle for defining recursive functions over the ADT.

Example 1. A single-linked list $\mathbb{L}$ over elements of sort $\mathbb{Z}$ is defined as *nil* (i.e., a base constructor) or *cons* (i.e., an inductive constructor that takes as input an integer, called the head, and another list, called the tail). Examples of recursive functions over lists include the length, append, and reverse. We use a mnemonic notation to represent lists as sequences of elements, i.e., $\langle 1, 2 \rangle$ stands for a list constructed by $cons(1, cons(2, nil))$.

For proving the validity of a formula $\forall x . \varphi(x)$, where variable x has sort s, we follow the well-known principle of *structural induction*. That is, we prove independently the base case (i.e., that $\varphi(bc_s)$ holds), then generate inductive hypotheses (i.e., formulas of form $\varphi(x_i)$ for fresh variables x_i, which correspond to sort s) and prove the inductive step, (i.e., that all $\varphi(x_1), \ldots, \varphi(x_n)$ imply $\varphi(ic_s(y_1, \ldots, y_k, x_1, \ldots, x_n, y_{k+1}, \ldots, y_m))$), where y_i has sort s_i.

Throughout the paper, we are interested in determining the validity of formulas of the form $\forall x.\varphi(x)$, where φ may have nested universal quantifiers:

$$\forall x.\Big(\forall y.\psi(y)\Big) \wedge \ldots \wedge \Big(\forall z.\gamma(z)\Big) \implies \theta(x) \tag{1}$$

Formulas $\psi, \ldots, \gamma$ on the left side of the implication (1) are called *assumptions*. If an assumption is not implied by any combination of other assumptions, it is called an *axiom* (otherwise, a *lemma*). The formula θ on the right side of the implication is called a *goal*.

Definition 2 (CHC). Assume that $\mathcal{X}$ is a countable set of variables associated with a sort $\mathcal{S}$. A first-order language $\mathcal{A}$ of quantifier-free formulas over $\mathcal{F}$, $\mathcal{P}$, and $\mathcal{X}$ is called a *constraint language*. A formula $\varphi \in \mathcal{A}$ is called a *constraint*. A *constrained Horn clause* (*CHC*) is a formula in first-order logic of the form:

$$\varphi \wedge r_1(x_1, \ldots, x_n) \wedge \ldots \wedge r_p(y_1, \ldots, y_k) \implies H$$

where we consider a fixed set $\mathcal{R}$ of *uninterpreted relation symbols*, such that $\mathcal{R} \cap (\mathcal{F} \cup \mathcal{P}) = \varnothing$. Expression H, called the *head* of the clause, is either an application $r_0(z_1, \ldots, z_m)$ or constant $\bot$. Each r_i is an uninterpreted relation symbol ($r_i \in \mathcal{R}$) and φ is a constraint. Each x_i, y_j, and z_k is a variable from $\mathcal{X}$.

The left side of a CHC C is called the *body*. If there are no symbols from $\mathcal{R}$ in the body of C, then C is called a *fact*. If the head of C is $\bot$, then it is called a *query*. Otherwise, C is called *inductive*. If an inductive CHC contains one application of an uninterpreted relation symbol in the body, it is called *linear*, and *non-linear*, if more than one.

Definition 3. Given a set S of CHCs and $r \in \mathcal{R}$, the *definitive rules* (denoted just *rules*) of r is a subset of S, such that:

$$\mathit{rules}(S, r) \overset{\text{def}}{=} \{C \in S \mid \mathit{head}(C) = r(\cdot)\}$$

Definition 4. A system of CHCs S over $\mathcal{R}$ is called satisfiable if there exists an *interpretation* $\mathcal{I}$ for each uninterpreted relation symbol from $\mathcal{R}$ in $\mathcal{A}$, that make all CHCs from S valid. It defines a *solution* of the system, also referred to as *inductive invariant*.

Technically Definition 4 needs substitutions defined as follows. Let φ be a formula, and $\mathcal{I}$ be an interpretation for $\mathcal{R}$. Then a substitution is the formula $\varphi[\mathcal{I}/\mathcal{R}]$ obtained from φ by replacing each occurrence of a formula of the form $r(x_1, \ldots, x_n)$ by $\mathcal{I}(r)(x_1, \ldots, x_n)$, where $r \in \mathcal{R}$. We naturally generalize this notation to sets of formulas (e.g., CHCs).

Systems of CHCs serve as compact representations of symbolic program encodings (i.e., for any number of loop iterations and any recursive depth). Automated verification is then reduced to determining the satisfiability of the corresponding systems of CHCs, and their solutions represent safe inductive invariants, i.e., formulas that over-approximate the sets of reachable states, but precise enough to prove unreachability of the error state. In this paper, we focus on finding recursive invariants for CHC systems, having no assumptions about the programming language used for writing the original program (thus, the approach can be used at the backends of verification tools such as [25,39]).

3 Recursive Functional Synthesis

The problem of *functional synthesis* (FS) is intuitively formulated as extracting a function implementation from its declarative specification. More formally, the problem is concerned with determining the satisfiability of a second-order formula $\exists f.\forall \vec{x}.p(f, \vec{x})$ over the uninterpreted function symbol f.

3.1 From CHC to FS

When it comes to representing declarative specification over ADTs, it is convenient to rely on CHCs over auxiliary uninterpreted predicates to represent unbounded computation over the structure of these ADTs. At the same time, if we assume that interpretations of predicates can only involve equalities and uninterpreted function symbols, then conjunctions of (universally quantified) CHCs can be rewritten to FS tasks that, if solved, provide solutions to the initial CHC tasks. In general, answering FS queries is difficult (undecidable), but if the structure of formulas is known, some successful heuristics can apply. In the rest of the section, we formulate a syntactic fragment and a tailored heuristic for solving FS problems over ADTs.

Definition 5. Given a set of CHCs S over a single relational symbol r, we say that a set of (universally quantified) first-order formulas $S_{f,\vec{x},y}$ is *CHC-inspired* if $S_{f,\vec{x},y} \stackrel{\text{def}}{=} \{c \mid c[r/\lambda\vec{x}, y \,.\, y = f(\vec{x})] \in S\}$.

That is, a CHC-inspired set of formulas can be constructed from CHCs after replacing all uninterpreted predicates by equalities involving uninterpreted functions. In the following, we assume we are given a set of CHCs S over some r, all the CHCs are *definitive* (recall Definition 3), and the are implicit syntax assumptions about the shape of CHCs (which we overcome in Sect. 3.2). Then, we show how we can mechanically construct a CHC-inspired set after some analysis and transformation of the CHCs.

Definition 6. Let,

- r be a relation with arity n,
- i be a natural number such that $i < n$,
- C be a set of constructors such that $T = \langle s, C\rangle$ is an ADT, and
- $D = \{B \implies r(a_1, ..., a_n)\}$ be a set of definitive CHCs.

We define C_i to be the set of i-th arguments of heads of CHCs in D. That is,

$$C_i = \{a_i \mid \big(B \implies r(a_1, ..., a_N)\big) \in D\}.$$

We say that the set D is $\langle T, r, i\rangle - \textit{complete}$ if:

1. $|C_i| = |C|$,
2. For each constructor $c \in C$ there is an element $c(\cdot) \in C_i$.

Note: this means there is a bijection between the set C_i and the set of constructors C for the ADT, $T = \langle s, C \rangle$.

We call such argument position i an *inductive input argument* position. Intuitively, a set of implications should have a representative for each constructor of some ADT among arguments of the head, and the argument position should be the same for all the implications. It is useful in the next phase of our functional synthesis procedure: since targeting the construction of recursive definitions, we need to separately construct the base and inductive cases to satisfy all constructors of the inductive input argument.

Example 2. The CHCs below over the relational symbol p represent the computation of the length and the sum of a linked list using a single traversal of the data structure.[1]

$$\begin{aligned} \ell = 0 \land s = 0 &\implies p(\mathit{nil}, \ell, s) \\ p(xs', \ell', s') \land \ell = \ell' + 1 \land s = s' + x &\implies p(\mathit{cons}(x, xs'), \ell, s) \end{aligned}$$

This set is a set of definitive CHCs since the head of each implication has a relation symbol. Now we define the set $C_1 = \{\mathit{nil}, \mathit{cons}(x, xs')\}$. Since C is defined as $C = \{\mathit{nil}, \mathit{cons}\}$ it is clear that $|C_1| = |C|$. Similarly it is clear to for all elements of C_0 there is a corresponding element in C. Thus we say that this set of CHCs is $\langle \mathbb{L}, p, 1 \rangle$-complete.

The corresponding CHC-inspired set $S_{f,xs,s}$ allows us to *embed* a recursive function for computing a sum into the relation p:

$$\begin{aligned} \ell = 0 \land s = 0 &\implies s = f(\mathit{nil}) \\ s' = f(xs') \land \ell = \ell' + 1 \land s = s' + x &\implies s = f(\mathit{cons}(x, xs')) \end{aligned} \tag{2}$$

Similarly, we can define another CHC-inspired set $S_{g,xs,\ell}$ for a g function for computing the length of the list. Functions f and g can then be discovered separately and they do not contradict each other in a sense that the conjunction $s = f(xs) \land \ell = g(xs)$ is an invariant for the initial CHC system. Solutions for the FS problems $\exists f \,.\, \bigwedge_i S_{f,xs,s}$ and $\exists g \,.\, \bigwedge_i S_{g,xs,\ell}$, respectively, are as follows:

$$\begin{aligned} f &= \lambda xs. \begin{cases} 0 & \text{if } xs = \mathit{nil} \\ f(xs') + x & \text{if } xs = \mathit{cons}(x, xs') \end{cases} \\ g &= \lambda xs. \begin{cases} 0 & \text{if } xs = \mathit{nil} \\ g(xs') + 1 & \text{if } xs = \mathit{cons}(x, xs') \end{cases} \end{aligned} \tag{3}$$

[1] Although it is conventional in practice to compute the length and the sum in different traversals, it is not necessarily more efficient to do it this way. Also, combining traversals might be needed in verification purposes, see e.g. [42].

3.2 The Eq-Prop Transformation

Our approach to recursive functional synthesis is driven by a transformation of the formulas originated from the given CHCs. The key idea is to ultimately rewrite the head by as many equalities from the body, aiming to produce a *recurrence relation*. We introduce an EQ-PROP transformation whose purpose is twofold: by moving constraints from left to right, we 1) aim at constructing an equality having two (or more) applications of the function symbol f (but with different arguments), and 2) normalize the given CHCs with respect to the Definition 6, thus facilitating the inductive input arguments detection.

More formally, if the head H has an instance of term b, and the body has equality $a = b$, then the transformation replaces b by a in H and removes $a = b$ from the body:

$$\frac{(a = b) \wedge C \implies H(b, \cdot)}{C \implies H(a, \cdot)} \text{ [EQ-PROP]}$$

Example 3. Applying EQ-PROP once to the formulas below would replace xs in the heads of both implications and yield (2):

$$\begin{aligned} xs = nil \wedge \ell = 0 \wedge s = 0 &\implies s = f(xs) \\ s' = f(xs') \wedge xs = cons(x, xs') \wedge \ell = \ell' + 1 \wedge s = s' + x &\implies s = f(xs) \end{aligned}$$

Further, applying EQ-PROP two more times sequentially replaces s and then s' in the heads, yielding:

$$\begin{aligned} \ell = 0 &\implies 0 = f(nil) \\ \ell = \ell' + 1 &\implies f(xs') + x = f(cons(x, xs')) \end{aligned}$$

The remaining formulas in the bodies are then removed by quantifier elimination, and the resulting recurrence relation for f (also, an interpretation for the function symbol) can be used to extract function interpretation (3).

Theorem 1. *Given a $\langle T, r, i \rangle$-complete set of CHCs S for some r, let $S_{f,xs,s}$ be its CHC-inspired set of formulas. If applying* EQ-PROP *(possibly, multiple times) to $S_{f,xs,s}$ yields a recurrence relation for f, then a solution for the functional synthesis problem $\exists f \,.\, \bigwedge_i S_{f,xs,s}$ can be extracted from the recurrence relation.*

The proof of the theorem follows from definitions and the soundness of EQ-PROP: by propagating equalities and rewriting the formula at the right, we essentially perform quantifier elimination, thus preserving the satisfiability of the formula. When checking the validity of the functional synthesis solution on the initial CHCs, the bodies of CHC will compensate for the equalities that were eliminated during EQ-PROP application. Lastly, Definition 6 guarantees that the constructed recurrence relation is well-formed and its branches do not contradict each other.

4 Recursive Invariants

Solving arbitrary CHCs over ADTs is challenging. Because of the unbounded nature of data structures, the creation, modification, or folding of them requires the introduction of multiple recursive functions, as well as proving inductive properties about them. In particular, CHC may not only represent function definitions, but also assumptions/assertions about data and reachability information.

Example 4. A CHC system below gives a number of constraints over the theory of $\mathbb{L}$:

$$\begin{aligned}
xs = nil &\implies \boldsymbol{app}(xs, ys, ys)\\
xs = cons(x, xs') \land zs = cons(x, zs') \land \boldsymbol{app}(xs', ys, zs') &\implies \boldsymbol{app}(xs, ys, zs)\\
\boldsymbol{app}(xs, ys, rs) \land \boldsymbol{app}(ys, zs, ts) \land \boldsymbol{app}(xs, ts, us) &\implies \boldsymbol{app}(rs, zs, us)\\
xs = nil &\implies \boldsymbol{rev}(xs, xs)\\
xs = cons(x, xs') \land \boldsymbol{rev}(xs', ys') \land \boldsymbol{app}(ys', cons(x, nil), ys) &\implies \boldsymbol{rev}(xs, ys)\\
\boldsymbol{rev}(xs, xs') \land \boldsymbol{rev}(ys, ys') \land \boldsymbol{app}(xs', ys', zs')\land &\\
\boldsymbol{app}(ys, xs, rs) \land \boldsymbol{rev}(rs, rs') \land \neg(zs' = rs') &\implies \bot
\end{aligned}$$

The first two CHCs define the append of one list xs to another list ys. The first CHC gives the base case of $\boldsymbol{app}$, i.e., if xs is empty then the result equals ys. (Technically, predicate $\boldsymbol{app}$ has arity three, and the first two arguments of each $\boldsymbol{app}(\cdot, \cdot, \cdot)$ represent inputs, and the last one represents the output. Note that the last two arguments in the head of the first CHC are the same, indicating that appending nil to any ys does not change ys.) The second CHC gives the inductive case of $\boldsymbol{app}$, i.e., to append some $cons(x, xs')$ to some ys, we first need to append xs' to ys and then to cons x to the result.

The third CHC gives an additional associativity-like constraint over $\boldsymbol{app}$. Note that this can be derived from the previous CHCs and does not affect the satisfiability of the CHC system. We give it mainly for the following reasons: 1) the provided CHC system is syntactically a feasible input to our algorithm, and it still needs to be handled, and 2) our algorithm is capable of separating such CHCs from the remaining definitive CHCs for $\boldsymbol{app}$ and further solving the FS problem.

The next two CHCs describe the process of reversing a list using $\boldsymbol{app}$. Again, the base rule applies to an empty list, and the inductive rule applies to some $cons(x, xs')$, i.e., xs' needs to be reversed first and then placed in the resulting list before x. Lastly, the query CHC gives a constraint on both $\boldsymbol{app}$ and $\boldsymbol{rev}$: given lists xs and ys, and their reverses xs' and ys', then appending xs' to ys' yields the same result as reversing the append of ys and xs.

We target the invariant generation via recursive functional synthesis. Note that this is different from a direct way of generating recursive invariants, i.e., where an interpretation of the predicate has applications of the same predicate (see e.g., [16]). Finding such interpretations, however, could be tricky in the cases of nonlinear or nested recursion. We propose to extract function definitions from CHCs and extend the syntax of the underlying constraint language for each CHC problem by these functions, thus allowing us to formulate invariants using the recursive functions.

Example 5. Recall Example 4. The CHC system is satisfied by the following invariants:

$$\begin{aligned}\boldsymbol{app} &\mapsto \lambda xs, ys, zs \,.\, zs = f_{\boldsymbol{app}}(xs, ys)\\ \boldsymbol{rev} &\mapsto \lambda xs, ys \,.\, ys = f_{\boldsymbol{rev}}(xs)\end{aligned}$$

where:

$$f_{\boldsymbol{app}} = \lambda xs, ys. \begin{cases} ys & \text{if } xs = nil\\ cons(x, f_{\boldsymbol{app}}(xs', ys)) & \text{if } xs = cons(x, xs')\end{cases}$$

$$f_{\boldsymbol{rev}} = \lambda xs. \begin{cases} nil & \text{if } xs = nil\\ f_{\boldsymbol{app}}(f_{\boldsymbol{rev}}(xs'), cons(x, nil)) & \text{if } xs = cons(x, xs')\end{cases}$$

5 Solving CHCs over ADTs

In this section we introduce our main contribution: an algorithm to solve CHCs over ADT using functional synthesis.

5.1 Challenges of Recursive Functional Synthesis When Dealing with Arbitrary CHCs

When dealing only with a subset of definitive rules, Recursive Functional Synthesis (RFS) is straightforward. In general, CHC solving may provide additional challenges due to 1) presence of multiple relations, 2) additionally provided lemmas (syntactically, in the form of definitive rules, but outside of any $\langle T, r, i\rangle$-complete sets) that need to be validated, and 3) queries that need to be validated.

However, the first obstacle for applying any RFS reasoning is the possible uncertainty when deciding which argument of an uninterpreted relation symbol should be picked as a return argument. We can however apply EQ-PROP multiple times and eliminate as many equalities from the body of a CHC as possible, and then proceed to guessing an equality such that EQ-PROP could be applied again. This is achieved by introducing a fresh uninterpreted *function symbol* and using it to replace an uninterpreted *relation symbol*, thus posing an RFS query. Rule NEW-FUN formulates precisely how this transformation is applied to a CHC with applications of predicate r in the body and in the head. Additionally, to guarantee that EQ-PROP can be applied afterwards, our algorithm picks a common subterm a_i (if it exists) among arguments of r:

$$\frac{[r(a_1, .., a_i, .., a_n) \wedge] B \implies r(b_1, .., b_i, .., b_n)}{[a_i = f_{r,i}(a_1, .., a_{i-1}, a_{i+1}, .., a_n) \wedge] B \implies b_i = f_{r,i}(b_1, .., b_{i-1}, b_{i+1}, .., b_n)}\ [\text{NEW-FUN}]$$

Here, $n = \mathit{arity}(r)$ and $f_{r,i}$ is a fresh symbol from $\mathcal{F}$. Note that this transformation does not guarantee success, i.e., it may make the system of constraints imposed by the CHC system unsatisfiable. However, the opposite claim is more optimistic: if after applying this transformation, all implications in the CHC system are valid, then the synthesized predicate interpretation:

$$\lambda x_1 \ldots x_n \,.\, x_i = f_{r,i}(x_1 \ldots x_{i-1}, x_{i+1}, \ldots x_n)$$

for some interpretation of $f_{r,i}$ can be used in the rest of the CHC solving process. In other words, after applying NEW-FUN, we update the current solution of the CHC system to map $I(r) = \lambda x_1 \ldots x_n \,.\, x_i = f_{r,i}(x_1 \ldots x_{i-1}, x_{i+1}, \ldots x_n)$, then apply EQ-PROP again and substitute the interpretations from I to predicates in the whole CHC system.

5.2 Core Algorithm

Our algorithm takes a system of CHCs S and a set of uninterpreted relation symbols $\mathcal{R}$ as input and determines the satisfiability of S. The main idea behind creating a recursive interpretation for an uninterpreted predicate is to replace it with an uninterpreted function, and then derive the *definition* for these functions. We require identifying the *inductive input argument* and the *return argument* among the arguments of the uninterpreted predicate. To well-define a recursive function, the inductive input argument should have the ADT sort, and we should select enough CHCs to cover functionality for the base and recursive cases.

The first step of the algorithm (line 1) is the ordering of uninterpreted predicates $\mathcal{R}$ so that if r_i will be processed before r_j, then r_i must not depend on r_j. This partial ordering enables us to use the already discovered interpretations of dependent predicate.

Definition 7 (Predicate Dependency Ordering). Let $\mathit{rules}(S, r)$ be as defined in Definition 3, given $r_i, r_j \in \mathcal{R}$, we say that r_i depends on r_j (written $r_i \prec r_j$) if:

- $r_i \neq r_j$, and
- $r_j \in \mathit{rules}(S, r_i)$, or there exists another $r_k \in \mathcal{R}$, such that $r_i \prec r_k$ and $r_k \prec r_j$, and
- $r_i \notin \mathit{rules}(S, r_j)$, and $\nexists\, r_k \in \mathcal{R}$ such that $r_j \prec r_k$ and $r_k \prec r_i$.

Example 6. The system of CHCs in Example 4 has two uninterpreted relation symbols, ***app*** and ***rev***, and the inductive CHC for ***rev*** applies ***app*** in the body, making $\boldsymbol{rev} \prec \boldsymbol{app}$. The algorithm thus finds an interpretation for ***app*** first and then proceeds to ***rev***.

If all predicates in the CHC system can be ordered, the algorithm then proceeds to synthesizing an implementation for every relation symbol, beginning with the ones that do not have any dependencies (line 2). At each iteration of this loop, the algorithm aims at synthesizing an interpretation for a single relation $r \in \mathcal{R}$. It maintains an invariant (line 4) that all relation symbols except

Algorithm 1: ADT-CHC: Solving CHCs over ADTs.

```
   Input: CHC system $S$, uninterpreted predicates symbols $\mathcal{R}$
   Output: $res \in \{\textsc{sat}, \textsc{unknown}\}$, interpretations $I$ for $\mathcal{R}$
 1 $order \leftarrow \textsc{OrderPredicates}(\mathcal{R})$;
 2 for $(r \leftarrow \textsc{top}(order); r \in order; r \leftarrow \textsc{next}(order))$ do
 3     $rules \leftarrow \{C \in S \mid head(C) = r(\cdot)\}$;
 4     assert $\forall r' \in \mathcal{R} \setminus \{r\}$ . if $r'$ is used in $rules$ then $I(r')$ is defined;
 5     $rules \leftarrow rules[I/\mathcal{R}]$;
 6     $\big(rules \leftarrow \textsc{eq-prop}(rules)\big)^*$;
 7     if $i \in \varnothing$ then
 8         return unknown;
 9     let $rules_T$ be a $\langle T, r, i\rangle$-complete subset of $rules$ for some $T$;
10     for $j \in [1, i) \cup (i, arity(r)]$ do
11         let $C'$ be the result of applying $\textsc{eq-prop} \circ \textsc{new-fun}_j$
                                       to some $C \in rules_T$ such that $C' \neq C$;
12         if $C' \in \varnothing$ then continue;
13         $sol \leftarrow \lambda x_1 \ldots x_n \,.\, x_j = f_{r,j}(x_1 \ldots x_{j-1}, x_{j+1}, \ldots x_n)$;
14         if $I(r)$ is not defined then $I(r) \leftarrow sol$;
15         else $I(r) \leftarrow \lambda x_1 \ldots x_n \,.\, I(r)(x_1 \ldots x_n) \wedge sol$;
16     $rules' \leftarrow rules[I(r)/r]$;
17     let $rules'_T$ be a $\langle T, r, i\rangle$-complete subset of $rules'$ for some $T$;
18     $D \leftarrow \bigwedge_{C \in rules'_T} \forall vars(C) \,.\, C$;
19     if $\textsc{isValid}(Lemmas \wedge D \implies rules')$ then
20         $Lemmas \leftarrow Lemmas \wedge D$;
21     else
22         return unknown;
23 assert $\forall r \in \mathcal{R} \implies I(r)$ is defined;
24 if $\textsc{isValid}\Big(Lemmas \implies \textsc{apply}\big(\{C \in S \mid head(C) = \bot\}, I\big)\Big)$ then
25     return sat;
26 return unknown;
```

r that occur in the definitive *rules* of r are already mapped to their interpretations, i.e., that all previous iterations of the loop succeeded. This enables us to use all interpretations (line 5): we simply replace all predicate symbols in all CHCs with the corresponding interpretations.

For the further processing of *rules* of r, we require the rules to deterministically identify the branches of the recursive function, denoted $f_{r,i}$, that corresponds to r. To precisely determine that, the algorithm first identifies the inductive input argument of $f_{r,i}$. It applies rule EQ-PROP (line 6 which uses the Kleene star notation to reflect the continuous nature of the rule application until a fixedpoint is reached) for every rule in *rules*. If no inductive input argument is found (line 7), the algorithm cannot proceed. Otherwise, it attempts to find a return argument.

The nested loop in lines 10–15 approaches various possible return arguments of r (i.e., excluding the inductive input arguments). It searches for an implication C if $rules_T$ where the composition of NEW-FUN and EQ-PROP successfully applies, i.e., the body permits a replacement of some relation symbol by a new equality. This gives a new interpretation of r as a conjunction of equalities over new function symbols $f_{r,i}$ (line 15) that is recorded in I. A definition of $f_{r,i}$ is created by rewriting this interpretation in all $rules_T$ (line 18) and universally quantifying all free variables. To check the correctness of the constructed definitions and interpretations for r, the algorithm uses all the remaining *rules* after the substitution and check their actual validity using a theorem prover (line 19).

Example 7. In order to confirm that the third CHC in $rules(S, \boldsymbol{app})$ is valid after the substitution of the interpretation that uses the definition of $f_{\boldsymbol{app}}$ in Example 5, we prove the validity of the following formula (which succeeds by induction on xs):

$$\begin{aligned}
&\forall ys\,.\,f_{\boldsymbol{app}}(nil, ys) = ys \land \\
&\forall xs, ys, x\,.\,f_{\boldsymbol{app}}(cons(x, xs), ys) = cons(x, f_{\boldsymbol{app}}(xs, ys)) \implies \\
&\qquad\qquad \forall xs, ys, rs, zs, ts, us, f_{\boldsymbol{app}}(xs, ys) = rs \land f_{\boldsymbol{app}}(ys, zs) = ts \land \\
&\qquad\qquad\qquad\qquad f_{\boldsymbol{app}}(xs, ts) = us \implies f_{\boldsymbol{app}}(rs, zs) = us)
\end{aligned}$$

Interestingly, this query can be recycled in the remainder of the algorithm to accelerate the solving process of the query.

A successful ending of the algorithm is when the theorem prover returns VALID for all the queries.

Theorem 2. *If the algorithm terminates with the* SAT *result (line 25), the input CHC system is satisfiable.*

The theorem can be proved by observing that the algorithm succeeds when all interpretations are found and a recursive function is synthesized for each predicate in $\mathcal{R}$. The soundness of interpretations with respect to intermediate goals is captured in the nested loop (line 19), and if the theorem prover does not succeed for some goal, the next possible return argument is considered. If no suitable return argument is found, then the algorithm does not find an interpretation: it either terminates with an UNKNOWN, or violates either of assertions in lines 4 or 23 (and thus, terminates with an UNKNOWN too).

Lastly, note that the backtracking in our pseudocode is simplified away for demonstration reasons. In fact, it could be the case that for a couple of relations $r_k \prec r_j$, there are two valid return arguments (or inductive input arguments) for an interpretation of r_j, but only one of them works for r_k. In this case, the algorithm needs to backtrack from processing r_k to r_j and re-synthesize the interpretation and the recursive function w.r.t. another argument(s). In our pseudocode, this can be simulated by running the algorithm again and making different decisions in lines 19, and/or 11, or being more selective in the loop in line 10. Evidently, in practice, it can be implemented in a more efficient way.

6 Automated Induction with AdtInd

In this section, we give an overview of our AdtInd prover that handles quantified formulas over ADT. It is specifically applied to prove the validity of formulas that arise at different stages of Algorithm 1, in this paper. However, it can also be used as a standalone tool and attempt user-given inputs.

6.1 Overview

The prover is a partial reimplementation of the work initially published in [59] and extended with new features. It follows the structural induction principle: it picks one quantified ADT-variable at a time and generates the base-case subgoal, inductive hypotheses, and the inductive-step subgoal. It then either uses an SMT solver to derive the subgoals directly from the assumptions (i.e., inductive hypotheses, recursive function definitions, or automatically generated lemmas), rewrites subgoals using the assumptions, or splits subgoals into a series of smaller subgoals to be solved recursively.

We refer the reader to the high-level presentation in [59] for a precise pseudocode. To simplify the presentation, we demonstrate the flow of AdtInd on a particular example of validating the synthesized interpretation on the query from Example 4.

Example 8. AdtInd begins with posing a quantified query and then simplifies it:

$$\begin{aligned}
\forall ys \,.\, f_{\boldsymbol{app}}(nil, ys) &= ys \wedge \\
\forall xs, ys, x \,.\, f_{\boldsymbol{app}}(cons(x, xs), ys) &= cons(x, f_{\boldsymbol{app}}(xs, ys)) \wedge \\
f_{\boldsymbol{rev}}(nil) &= nil \wedge \\
\forall xs, x \,.\, f_{\boldsymbol{rev}}(cons(x, xs)) &= f_{\boldsymbol{app}}(f_{\boldsymbol{rev}}(xs), cons(x, nil)) \implies \\
&\forall xs, ys \,.\, f_{\boldsymbol{rev}}(f_{\boldsymbol{app}}(xs, ys)) = f_{\boldsymbol{app}}(f_{\boldsymbol{rev}}(ys), f_{\boldsymbol{rev}}(xs))
\end{aligned}$$

Structurally, the formula above is a logical implication, and the conjunction on its left consists of recursive definitions of $f_{\boldsymbol{app}}$ and $f_{\boldsymbol{rev}}$. These are the universally quantified formulas that initially form the set of assumptions. Further, on the right of the formula, there are two quantified ADT variables xs and ys, and AdtInd initiates a proof by induction over one of them, xs. The base case is just:

$$\forall ys \,.\, f_{\boldsymbol{rev}}(f_{\boldsymbol{app}}(nil, ys)) = f_{\boldsymbol{app}}(f_{\boldsymbol{rev}}(ys), f_{\boldsymbol{rev}}(nil))$$

After rewriting the base cases of the definitions of $f_{\boldsymbol{app}}$ and $f_{\boldsymbol{rev}}$, the goal becomes:

$$\forall ys \,.\, f_{\boldsymbol{rev}}(ys) = f_{\boldsymbol{app}}(f_{\boldsymbol{rev}}(ys), nil)$$

The description of the steps to prove it is deferred to Example 9. Then, AdtInd generates an inductive hypothesis for a fixed xs that is added to the assumptions:

$$\forall ys \,.\, f_{\boldsymbol{rev}}(f_{\boldsymbol{app}}(xs, ys)) = f_{\boldsymbol{app}}(f_{\boldsymbol{rev}}(ys), f_{\boldsymbol{rev}}(xs))$$

and formulates a new subgoal over the same xs which is further proved valid:

$$\forall ys, x \,.\, f_{rev}(f_{app}(cons(x, xs), ys)) = f_{app}(f_{rev}(ys), f_{rev}(cons(x, xs))).$$

Two important features that AdtInd relies on are *lemma generation* (see Sect. 6.2) and *filtering lemma candidates* (Sect. 6.3). Both of them are designed for situations when a current subgoal does not immediately follow from the current assumptions (e.g., it may require a proof by induction). Our strategy is to synthesize a lemma, the validity of which is substantially easier to be proved than the validity of the goal. However, during the synthesis, we may end up with invalid lemma candidates. In this case, our approach leverages a filtering procedure that helps to remove some lemma candidates quickly.

6.2 Extracting Common Subterms for Helper Lemmas

Our approach generates auxiliary lemmas by replacing common subterms in the subgoal by fresh variables. An important condition for soundness of this method is that such newly introduced lemmas should themselves follow from the given assumptions. In particular, AdtInd separates the failure formula from the context, universally quantifies the variables, picks a subset of assumptions, and initializes the new solving process. If succeeded, this newly discovered assumption is added to the set of existing assumptions, and the solving process of the initial formula resumes.

Recall our motivating example. We demonstrate how the query can be proved using the recursive definitions of f_{app} and f_{rev}.

Example 9. Recall Example 8 and the following subgoal:

$$\forall ys \,.\, f_{rev}(ys) = f_{app}(f_{rev}(ys), nil)$$

At this point, AdtInd needs a helper lemma that can be discovered by proving the current goal by induction. However, the presence of f_{rev} unnecessarily complicates the process. In this case, AdtInd finds a common subterm, $f_{rev}(ys)$, replaces it by a fresh quantified variable and gets a new goal which is easily provable by induction:

$$\forall zs \,.\, zs = f_{app}(zs, nil).$$

AdtInd then adds this quantified formula as a new assumption, and the restarted proof process immediately concludes that this assumption implies the base case.

AdtInd has a systematic way for finding helper lemmas, demonstrated in the example above. Specifically, at the point when no assumption is applicable, our solver explores the parse tree of the goal and finds common patterns. Of particular interest are the applications of the same functions to the same tuples of arguments, as well as arithmetic and Boolean constraints. AdtInd then ranks them (more common patterns are considered first) and attempts to prove them one-by-one until either something is proved, or everything tried. In the latter case, the solver performs backtracking.

6.3 Filtering Procedure

As shown in the previous subsection, AdtInd implements a procedure to expand its own search space by generating helper lemmas from a set of candidate expressions. Each of these candidate formulas should be verified by an instance of AdtInd before it can be used. However, AdtInd is not designed to deal with invalid formulas, i.e., the ones for which no proving strategy could succeed, and thus AdtInd diverges. To resolve this problem, we present a filtering procedure that we call Disproof which quickly tries to filter *potentially invalid* lemmas.

The filtering procedure begins by enumerating ADT literals up to a certain depth. Then, the filtering procedure substitutes each quantified variable in the current goal for ADT literals to create a set of quantifier-free formulas. Finally, the filtering procedure rewrites the quantifier-free formulas to eliminate functions and sends the negations of the resulting formulas to an SMT solver. The Disproof procedure returns filter when the SMT solver finds at least one satisfiable negation, and thus the candidate formula is not considered in the process any longer. Otherwise Disproof returns unknown, and AdtInd attempts to prove it. Note that the procedure may filter a valid candidate formula that could be potentially useful. However since we use this procedure only to accelerate the search of lemmas (i.e., whenever a candidate lemma is filtered, AdtInd quickly jumps to another candidate), the soundness of the entire procedure is not compromised.

Algorithm 2 gives a pseudocode of this procedure. It receives a formula $\forall x_1, \ldots, x_n \,.\; G(x_1, \ldots, x_n)$ over n universally quantified ADT variables. The algorithm begins with generating n sets of ADT literals (line 4) for each of the ADT variables $x_1, \ldots, x_n$, where each literal has depth at most k. Intuitively, this procedure is recursive:

- At level 0, ADTLitGen$(x_i, 0)$ returns a singleton set T_0 consisting of an application of the base constructor of variable x_i.
- At level k, it assumes a set T_{k-1} is generated for level $k-1$. Then, for the inductive constructor ic for the sort of x_i with arity m that uses p arguments of the sort of x_i, and each subset of p literals $\ell_1, \ldots, \ell_p \in T_{k-1}$, ADTLit-Gen$(x_i, k)$ generates $m-p$ fresh variables $v_1, \ldots, v_{m-p}$ and applies ic to $\ell_1, \ldots, \ell_p, v_1, \ldots, v_{m-p}$. The resulting literal is added to T_k.

Importantly, ADTLitGen does not generate concrete literals, except of the one at level 0. We let the literals to use fresh variables and use an SMT solver to evaluate them, if possible, such that the resulting concrete literals violate the goal.

The algorithm further substitutes each combination of the generated ADT literals for all $x_1, \ldots, x_n$ in G (line 6) and proceeds to rewriting the resulting formula using the given assumptions. For each substitution, the algorithm performs rewriting using assumptions until no more rewrites are possible (for more information, see [59]). Finally, if a rewritten term does not have occurrences of any functions or predicates defined in the assumptions, then its negation can be checked for the satisfiability with an SMT solver (line 9). If it is satisfiable, then

Algorithm 2: Disproof: Fitering candidate expressions.

```
   Input: candidate expression of form $\forall x_1, \ldots, x_n . G(x_1, \ldots x_n)$,
   set of assumptions $A$, exploration depth: $k$
   Output: $res \in \{\text{FILTER}, \text{UNKNOWN}\}$
 1 for $i \in [1, n]$ do
 2     $lits_i \leftarrow \varnothing$;
 3     for $j \in [0, k]$ do
 4         $lits_i \leftarrow lits_i \cup$ ADTLitGen$(x_i, j)$;
 5 for $\langle \ell_1, \ldots, \ell_n \rangle \in lits_1 \times \ldots \times lits_n$ do
 6     $t \leftarrow G[\ell_1/x_1, \ldots \ell_n/x_n]$;
 7     $\big(t \leftarrow \text{REWRITE}(t, A)\big)^*$;
 8     if $t$ has occurrences only of constructors and equality then
 9         if isSAT$(\neg t)$ then
10             return FILTER;
11 return UNKNOWN
```

the violation of goal G is found (and a concrete ADT literal is extracted from the model generated by the solver for the variables we introduced).

Example 10. Suppose AdtInd generates the following helper lemma with f_{rev} defined as in previous examples:

$$\forall x . f_{rev}(x) = x$$

The filtering procedure then instantiates variable x with three ADT literals, enumerated up to depth 2, where v_0 and v_1 are fresh variables: *nil*, $cons(v_0, nil)$, and $cons(v_1, cons(v_0, nil))$, resulting in the following list of formulas:

$$\begin{aligned} f_{rev}(nil) &= nil \\ f_{rev}(cons(v_0, nil)) &= cons(v_0, nil) \\ f_{rev}(cons(v_1, cons(v_0, nil))) &= cons(v_1, cons(v_0, nil)) \end{aligned}$$

Next it unrolls each of these formulas by applying the definition of f_{rev}, resulting in:

$$\begin{aligned} nil &= nil \\ cons(v_0, nil) &= cons(v_0, nil) \\ cons(v_0, cons(v_1, nil)) &= cons(v_1, cons(v_0, nil)) \end{aligned}$$

Finally, our procedure tests the negation of each of these terms using an SMT solver and determines that for $v_0 \mapsto 0$ and $v_1 \mapsto 1$, the negation of the last equality is true. The procedure then returns FILTER, and AdtInd jumps to another candidate.

Note that when doing filtering the procedure does not send any assumptions to the solver, and the constructors are treated as uninterpreted functions. Without any extra axiomatization, some of the solver's SAT results might be spurious.

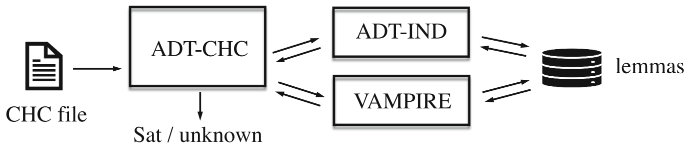

Fig. 1. CHC solving process with AdtChc with AdtInd or Vampire at the backend.

In principle, our procedure can be extended to become a sound refutation procedure, if a sufficient number of constraints about constructors are supplied. In our application, we trade precision for the speed of lemma generation, so even if we miss some potentially useful lemmas, the procedure continues with the next candidates and still has a chance to prove the main goal valid.

7 Implementation and Evaluation

In this section we present the overview of the implemented CHC solver and provide its evaluation compared to state-of-the-art.

7.1 Framework

Figure 1 gives an overview of the flow of the solving process. The tool takes a CHC file (in the conventional SMT-LIBv2 format) as input. In addition to ADTs, the inputs may have constraints over Linear Integer Arithmetic (LIA). During the solving process, AdtChc tightly communicates with its backend solvers, AdtInd (as was described in Sect. 5.2) and Vampire [36]. While posing ADT queries and receiving the confirmations of their validity, AdtChc converts a subset of CHCs to recursive functions and makes their definitions available for future use.

While solving for the validity, AdtInd relies on the recursive definitions of functions over ADT produced by AdtChc and successfully proved queries, and AdtInd automatically generates some helper lemmas on the fly. Lemmas are then shared among AdtChc and backend solver and can be observed by the user. On the lower level, AdtInd reduces the reasoning over ADTs to equality and uninterpreted functions (EUF), and uses the Z3 SMT solver [12] to discharge auxiliary formulas. Vampire, to the best of our knowledge, has its own satisfiability and theory solvers and it uses a portfolio approach for solving formulas.

The source code of AdtChc and its benchmarks are available at https://github.com/grigoryfedyukovich/aeval/tree/adt-chc.

7.2 Experiments

We have considered publicly available CHC benchmarks that encode well-known verification problems. Our ultimate goal is to find invariants to prove safety properties in these benchmarks. Because many tools are not designed to recursive invariant synthesis, they have a hard time to solve benchmarks. But we show that AdtChc is effective and outperforms the competitors on many benchmarks.

We have compared AdtChc to state-of-the-art CHC solvers participated in the CHC-COMP [50], namely Eldarica[2] [24], PCSat[3] [52], HOICE [8], and Racer[4] [27] – the extension of GSpacer [37]. We considered three sets of benchmarks in the CHC format complying with the CHC-COMP rules converted to the CHC format from benchmarks used by various theorem provers: the first set, contains 28 problems, is derived from benchmarks for AdtInd [59], the second one with 17 problems comes from CLAM [26], and the last with 26 problems is taken from LEON [56]. The safety verification properties are concerned about the correctness of various operations on lists, amortized queues and binary trees.

We configured AdtChc to run in four different modes (that correspond to four first columns of Table 1 and Table 2): the first two use AdtInd as the backend solver, and the last two use Vampire as the backend solver. In the first configuration of AdtInd (denoted **w/A(1)**), the backend solver performs exactly as described in Sect. 6 but without the candidate filtering method, so it tries to prove valid *all candidate lemmas* that are generated. In the second configuration (denoted **w/A(2)**), we added the candidate filtering method, and thus, the solver finds *potentially invalid* lemmas first and skips them (if the filtering is unsuccessful, then the solver tries to prove the candidate). It allows the solver to save time and proceed to discovery of new lemmas. For Vampire, in its first configuration (denoted **w/V(1)**) we use the default setting, and in the second configuration of Vampire (denoted **w/V(2)**), we force it conduct the proofs by structural induction.

We used a timeout of 300 s CPU time for each tool and configuration. Overall, there are 71 benchmarks, and 41 of them were solved by either the configuration of AdtChc +AdtInd. 42 benchmarks were solved by either the configuration of AdtChc +Vampire. More importantly, in total by either of four configurations of AdtChc, our tool solved 52 benchmarks. Among them, 31 benchmarks were not solved by any other other competing tool. Eldarica solved 24 benchmarks, Racer solved 16 and PCSat solved 9, and HOICE solved 19.

Comparing backends of AdtChc, it is apparent that AdtInd is on average significantly faster than Vampire. It could be attributed to the fact that the latter uses the portfolio mode. However, both tools have they strengths since there are benchmarks solely solved only with AdtInd and only with Vampire.

For benchmarking, we used a workstation equipped with 2.8 GHz Intel Core i7 4-Core (11th generation) and 12 GB of DDR4 RAM running Ubuntu 21.04.

[2] version 2.0.6.
[3] https://github.com/hiroshi-unno/coar.
[4] https://github.com/hgvk94/z3/tree/racer.

Table 1. Results (sec); "—" stands for "unknown".

	AdtChc							
Benchmark	w/A(1)	w/A(2)	w/V(1)	w/V(2)	Eldarica	Racer	PCSat	HOICE
ADTIND/heap_size	0.65	0.62	84.12	80.62	1.31	0.01	0.89	0.09
ADTIND/list_append_ass	0.48	0.4	119.93	0.06	—	—	—	—
ADTIND/list_append_len	1.1	1.03	120.5	0.1	24.1	—	—	—
ADTIND/list_append_min	1.97	2.3	120.47	120.45	—	—	—	—
ADTIND/list_append_min2	0.55	0.5	60.31	60.29	—	—	—	—
ADTIND/list_append_nil	0.71	0.67	60.28	0.06	1.38	0.02	—	0.1
ADTIND/list_append_sum	0.64	0.58	74.63	0.11	—	—	—	0.13
ADTIND/list_interleave	—	—	60.22	60.22	—	—	—	0.22
ADTIND/list_len_butlast	0.57	0.51	60.3	60.29	4.81	—	—	—
ADTIND/list_len_stren	4.09	5.53	—	—	1.36	0.01	1.21	1.12
ADTIND/list_len	0.35	0.31	66.32	0.08	1.11	0.02	0.96	0.08
ADTIND/list_min_max	0.76	0.72	60.31	15.91	1.66	0.02	1.53	—
ADTIND/list_min_sum_len	3.22	3.28	66.65	15.71	1.52	0.02	1.57	—
ADTIND/list_min_sum	1.72	1.75	74.74	73.89	—	—	—	—
ADTIND/list_rev_append	6.36	—	284.61	282.75	—	—	—	0.01
ADTIND/list_rev_len	1.87	1.69	126.6	66.12	—	—	—	—
ADTIND/list_rev	—	—	—	—	—	—	—	—
ADTIND/list_rev2_append	0.95	0.86	—	120.44	—	—	—	—
ADTIND/list_rev2_len	—	—	120.7	120.75	—	—	—	—
ADTIND/queue_amort	137.05	—	—	—	1.15	0.02	1.3	0.15
ADTIND/queue_len	—	—	—	—	—	—	—	—
ADTIND/queue_popback	—	—	—	—	49.11	—	—	—
ADTIND/queue_push_to_list	—	—	—	—	—	—	—	—
ADTIND/queue_push	—	—	—	—	—	—	—	—
ADTIND/tree_insert_all_size	3.22	3.18	217.8	213.72	—	—	—	—
ADTIND/tree_insert_size	0.33	0.25	—	—	—	—	—	—
ADTIND/tree_insert_sum	0.34	0.26	—	—	—	—	—	—
ADTIND/tree_size	0.64	0.58	77.69	0.1	1	0.01	0.78	0.07
CLAM/goal10	—	—	238.76	239.38	—	—	—	—
CLAM/goal11	—	—	240.36	239.98	—	—	—	—
CLAM/goal12	—	5.06	120.64	119.92	—	—	—	—
CLAM/goal17	—	—	238.32	239.82	—	—	—	—
CLAM/goal18	—	—	247.09	243.21	—	—	—	—
CLAM/goal19	—	—	238.4	239.56	—	—	—	—
CLAM/goal2	74.49	9.45	127.41	1.19	2.04	—	—	—
CLAM/goal21	—	—	—	—	—	—	—	—
CLAM/goal27	—	—	241.2	240.97	—	—	—	—
CLAM/goal3	1.29	1.02	120.52	0.12	—	—	—	—
CLAM/goal4	—	—	124.06	123.77	8.67	—	—	—
CLAM/goal5	—	5.2	126.74	126.58	—	—	—	—
CLAM/goal6	—	—	241.15	241.06	—	—	—	—
CLAM/goal7	2.76	2.7	120.78	120.75	—	—	—	—
CLAM/goal72	0.49	0.41	60.3	0.06	—	—	—	—
CLAM/goal8	—	31.15	206.28	205.47	—	—	—	—
CLAM/goal9	—	31.32	205.37	204.55	—	—	—	—
LEON/amortize-queue-goal1	0.6	0.54	66.28	0.11	—	—	—	—
LEON/amortize-queue-goal10	0.93	0.81	186.23	185.46	—	—	—	—
LEON/amortize-queue-goal11	0.47	0.41	120.38	60.2	—	—	—	—
LEON/amortize-queue-goal12	—	—	—	—	—	—	—	—
LEON/amortize-queue-goal13	—	—	—	—	—	—	—	—
LEON/amortize-queue-goal14	—	—	—	—	—	—	—	—
LEON/amortize-queue-goal15	—	—	—	—	—	—	—	—
LEON/amortize-queue-goal3	—	—	—	—	—	—	—	—
LEON/amortize-queue-goal4	—	—	—	—	2.33	—	—	0
LEON/amortize-queue-goal5	—	—	—	—	4.14	—	—	—
LEON/amortize-queue-goal6	—	—	—	—	—	—	—	0
LEON/amortize-queue-goal8	0.8	0.71	120.43	60.18	1.49	0.01	—	0.08
LEON/amortize-queue-goal9	—	—	—	—	—	—	—	—

Table 2. Results (cont).

Benchmark	AdtChc w/A(1)	AdtChc w/A(2)	AdtChc w/V(1)	AdtChc w/V(2)	Eldarica	Racer	PCSat	HOICE
LEON/bsearch-tree-goal1	198.64	—	—	—	—	—	—	—
LEON/bsearch-tree-goal10	—	—	—	—	2.81	0.05	—	0.29
LEON/bsearch-tree-goal11	67.1	—	—	—	0.84	0.01	—	0.52
LEON/bsearch-tree-goal12	17.92	15.56	—	—	1.01	0.01	—	0.29
LEON/bsearch-tree-goal13	30.73	34.32	—	180.62	—	—	—	—
LEON/bsearch-tree-goal14	—	—	—	—	0.56	0	0.91	0.09
LEON/bsearch-tree-goal2	—	127.23	271.82	263.31	—	—	—	—
LEON/bsearch-tree-goal3	—	—	278.39	281.68	—	—	—	—
LEON/bsearch-tree-goal4	—	—	—	—	—	—	—	—
LEON/bsearch-tree-goal5	—	—	—	—	—	—	—	—
LEON/bsearch-tree-goal6	—	273.95	—	—	0.94	0.01	—	0.4
LEON/bsearch-tree-goal8	89.45	—	—	—	0.63	0	0.79	0.07
LEON/bsearch-tree-goal9	—	280.87	—	—	0.66	0.01	—	0.23

8 Related Work

Existing CHC solvers [2,5,8,9,13,19,24,27–29,34,38,40,41,44,51,57,61] are utilized by the software model checkers for imperative languages [20,22], object-oriented languages [30,31], dataflow languages [17,18], and functional programming languages [7,15,33,43,58]. Algorithmically, solvers are based on Counterexample-Guided Abstraction Refinement (CEGAR) [10], Counterexample-Guided Inductive Synthesis (CEGIS) [52,54], Property Directed Reachability (PDR) [6,14], Machine Learning [53], but currently, there is no clear witness that any of these approaches are, in general, better than others. Furthermore, with the exception of [24], solvers are limited to relatively lightweight SMT theories and not ADT.

There is a plethora of proposed quantifier elimination algorithms and decision procedures for the first-order ADT fragment [3,45–47,55] and for an extension of ADT with constraints on term sizes [60]. As often useful for solving, the Craig interpolation procedure for ADT constraints has been proposed by [23]. Such techniques are being incorporated by various SMT solvers, like Z3 [12], CVC4 [4], and Princess [49]. Our SMT-based approach to handling ADTs uses a new functional synthesis approach: it works by rewriting CHCs and obtaining new definition from declarative CHC constraints. Lastly, there are approaches for CHC-based relational verification over ADTs [11,42] that effectively reduce reasoning to CHCs over lightweight SMT theories. These approaches do not generate inductive invariants over ADTs, while our approach does.

There is an approach where inductive invariants are represented by finite tree automata implemented in RingGen [35]. A system of CHCs over ADTs is rewritten into a formula over uninterpreted function symbols by eliminating all disequalities, testers, and selectors from the clause bodies. Then they reduce the satisfiability modulo theory of ADTs to satisfiability modulo EUF and apply off-the-shelf finite model finder to build a finite model of the reduced verification conditions. The automaton representing the safe inductive invariant are derived using the correspondence between finite models and tree automata. Unfortunately, RingGen works only on pure ADT (i.e., it defines natural numbers inductively as zero and +1, but we make use of Presburger arithmetic).

9 Conclusion and Future Work

We have presented a new approach to solve CHC problems over ADT using recursive function synthesis. Instead of generating recursive predicates, the approach generates recursive functions by applying semantics-preserving transformations to a subset of given CHCs determined on the fly. The remaining CHCs are used to validate the solutions and the approach reduces this problem to an off-the-shelf theorem prover that is expected to prove the validity of each universally quantified formula following the principle of structural induction. Our implementation called AdtChc exploits the Z3 SMT solver to process a number of quantifier-free queries over arithmetic, uninterpreted functions, and arrays. While AdtChc outputs a number of recursive definitions of functions that are used in interpretations of predicates, theorem provers AdtInd and Vampire automatically discharge the validity checks, often generating a number of useful lemmas that can be exchanged among queries and accelerate the solving process. We also presented two new features of AdtInd that help in its proving process: generation of helper lemmas from common subterms and filtering potentially useless candidate lemmas. We experimentally compared our tools with state-of-the-art, and it shows promising results. In the future, we plan to extend the set of features of the tools, and in particular support solving queries with nested (and possibly, alternating) quantifiers.

Acknowledgments. The work is supported in parts by a gift from Amazon Web Services.

References

1. Alur, R., et al.: Syntax-guided synthesis. In: FMCAD, pp. 1–17. IEEE (2013)
2. Bakhirkin, A., Monniaux, D.: Combining forward and backward abstract interpretation of horn clauses. In: Ranzato, F. (ed.) SAS 2017. LNCS, vol. 10422, pp. 23–45. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66706-5_2
3. Barrett, C., Shikanian, I., Tinelli, C.: An abstract decision procedure for a theory of inductive data types. J. Satisfiability, Boolean Model. Comput. **3**, 21–46 (2007)
4. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
5. Beyene, T.A., Popeea, C., Rybalchenko, A.: Solving existentially quantified horn clauses. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 869–882. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_61
6. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_7

7. Champion, A., Chiba, T., Kobayashi, N., Sato, R.: ICE-based refinement type discovery for higher-order functional programs. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 365–384. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_20
8. Champion, A., Kobayashi, N., Sato, R.: HoIce: an ICE-based non-linear horn clause solver. In: Ryu, S. (ed.) APLAS 2018. LNCS, vol. 11275, pp. 146–156. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-02768-1_8
9. Chen, Y.-F., Hsieh, C., Tsai, M.-H., Wang, B.-Y., Wang, F.: Verifying recursive programs using intraprocedural analyzers. In: Müller-Olm, M., Seidl, H. (eds.) SAS 2014. LNCS, vol. 8723, pp. 118–133. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10936-7_8
10. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_15
11. De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: Solving horn clauses on inductive data types without induction. TPLP **18**(3–4), 452–469 (2018)
12. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
13. Dietsch, D., Heizmann, M., Hoenicke, J., Nutz, A., Podelski, A.: Ultimate TreeAutomizer. In: HCVS/PERR, vol. 296 of EPTCS, pp. 42–47 (2019)
14. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: FMCAD, pp. 125–134. IEEE (2011)
15. Fedyukovich, G., Ahmad, M.B.S., Bodík, R.: Gradual synthesis for static parallelization of single-pass array-processing programs. In: PLDI, pp. 572–585. ACM (2017)
16. Fedyukovich, G., Ernst, G.: Bridging arrays and ADTs in recursive proofs. In: TACAS 2021. LNCS, vol. 12652, pp. 24–42. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72013-1_2
17. Garoche, P., Gurfinkel, A., Kahsai, T.: Synthesizing modular invariants for synchronous code. In: HCVS, vol. 169 of EPTCS, pp. 19–30 (2014)
18. Garoche, P. Kahsai, T., Thirioux, X.: Hierarchical state machines as modular horn clauses. In: HCVS, vol. 219 of EPTCS, pp. 15–28 (2016)
19. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: PLDI, pp. 405–416. ACM (2012)
20. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The seahorn verification framework. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 343–361. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_20
21. Hoare, C.A.R.: Recursive data structures. Int. J. Parallel Program. **4**(2), 105–132 (1975)
22. Hojjat, H., Konečný, F., Garnier, F., Iosif, R., Kuncak, V., Rümmer, P.: A verification toolkit for numerical transition systems. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 247–251. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_21
23. Hojjat, H., Rümmer, P.: Deciding and interpolating algebraic data types by reduction. In: SYNASC, pp. 145–152. IEEE (2017)
24. Hojjat, H., Rümmer, P.: The ELDARICA horn solver. In: FMCAD, pp. 158–164. IEEE (2018)

25. Hojjat, H., Rümmer, P., Shamakhi, A.: On strings in software model checking. In: Lin, A.W. (ed.) APLAS 2019. LNCS, vol. 11893, pp. 19–30. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-34175-6_2
26. Ireland, A., Bundy, A.: Productive use of failure in inductive proof. In: Zhang, H. (ed.) Automated Mathematical Induction, pp. 79–111. Springer, Cham (1996). https://doi.org/10.1007/978-94-009-1675-3_3
27. Hari Govind, V.K., Shoham, S., Gurfinkel, A.: Solving constrained horn clauses modulo algebraic data types and recursive functions. Proc. ACM Program. Lang. **6**(POPL), 1–29 (2022)
28. Kafle, B., Gallagher, J.P., Ganty, P.: Solving non-linear Horn clauses using a linear Horn clause solver. In: HCVS, vol. 219 of EPTCS, pp. 33–48 (2016)
29. Kafle, B., Gallagher, J.P., Morales, J.F.: Rahft: a tool for verifying horn clauses using abstract interpretation and finite tree automata. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 261–268. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_14
30. Kahsai, T., Kersten, R., Rümmer, P., Schäf, M.: Quantified heap invariants for object-oriented programs. In: LPAR, vol. 46 of EPiC Series in Computing, pp. 368–384. EasyChair (2017)
31. Kahsai, T., Rümmer, P., Sanchez, H., Schäf, M.: JayHorn: a framework for verifying java programs. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 352–358. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_19
32. Kim, J., Hu, Q., D'Antoni, L., Reps, T.: Semantics-guided synthesis. Proc. ACM on Program. Lang. **5**(POPL), 1–32 (2021)
33. Kobayashi, N., Sato, R., Unno, H.: Predicate abstraction and CEGAR for higher-order model checking. In: ACM, pp. 222–233. ACM (2011)
34. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 17–34. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_2
35. Kostyukov, Y., Mordvinov, D., Fedyukovich, G.: Beyond the elementary representations of program invariants over algebraic data types. In: PLDI, pp. 451–465 (2021)
36. Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 1–35. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_1
37. Vediramana Krishnan, H.G., Chen, Y.T., Shoham, S., Gurfinkel, A.: Global guidance for local generalization in model checking. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 101–125. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_7
38. Krishnan, H.G.V., Fedyukovich, G., Gurfinkel, A.: Word level property directed reachability. In: ICCAD, pp. 1–9. IEEE (2020)
39. Matsushita, Y., Tsukada, T., Kobayashi, N.: RustHorn: CHC-based verification for rust programs. In: ESOP 2020. LNCS, vol. 12075, pp. 484–514. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-44914-8_18
40. McMillan, K.L.: Lazy annotation revisited. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 243–259. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_16
41. McMillan, K.L., Rybalchenko, A.: Solving constrained Horn clauses using interpolation. In Technical report MSR-TR-2013-6 (2013)
42. Mordvinov, D., Fedyukovich, G.: Synchronizing constrained horn clauses. In: LPAR, vol. 46 of EPiC Series in Computing, pp. 338–355. EasyChair (2017)

43. Mordvinov, D., Fedyukovich, G.: Verifying safety of functional programs with rosette/unbound. CoRR, abs/1704.04558 (2017). https://github.com/dvvrd/rosette
44. Mordvinov, D., Fedyukovich, G.: Property directed inference of relational invariants. In: FMCAD, pp. 152–160. IEEE (2019)
45. Oppen, D.C.: Reasoning about recursively defined data structures. J. ACM (JACM) **27**(3), 403–411 (1980)
46. Pham, T., Gacek, A., Whalen, M.W.: Reasoning about algebraic data types with abstractions. J. Autom. Reason. **57**(4), 281–318 (2016)
47. Reynolds, A., Blanchette, J.C.: A decision procedure for (co) datatypes in SMT solvers. J. Autom. Reason. **58**(3), 341–362 (2017)
48. Reynolds, A., Kuncak, V.: Induction for SMT solvers. In: D'Souza, D., Lal, A., Larsen, K.G. (eds.) VMCAI 2015. LNCS, vol. 8931, pp. 80–98. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46081-8_5
49. Rümmer, P.: A constraint sequent calculus for first-order logic with linear integer arithmetic. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 274–289. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89439-1_20
50. Fedyukovich, G., Rümmer, P.: Competition report: CHC-COMP-21. In: Hojjat, H., Kafle, B. (eds.) Proceedings 8th Workshop on Horn Clauses for Verification and Synthesis, HCVS@ETAPS 2021, Virtual, 28th March 2021. EPTCS, vol. 344, pp. 91–108 (2021). https://doi.org/10.4204/EPTCS.344.7
51. Rümmer, P., Hojjat, H., Kuncak, V.: Disjunctive interpolants for horn-clause verification. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 347–363. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_24
52. Satake, Y., Unno, H., Yanagi, H.: Probabilistic inference for predicate constraint satisfaction. In: AAAI, pp. 1644–1651. AAAI Press (2020)
53. Sharma, R., Aiken, A.: From invariant checking to invariant inference using randomized search. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 88–105. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_6
54. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: ASPLOS, pp. 404–415. ACM (2006)
55. Suter, P., Dotta, M., Kuncak, V.: Decision procedures for algebraic data types with abstractions. ACM Sigplan Not. **45**(1), 199–210 (2010)
56. Suter, P., Köksal, A.S., Kuncak, V.: Satisfiability modulo recursive programs. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 298–315. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23702-7_23
57. Unno, H., Terauchi, T.: Inferring simple solutions to recursion-free horn clauses via sampling. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 149–163. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_10
58. Unno, H., Torii, S., Sakamoto, H.: Automating induction for solving horn clauses. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 571–591. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_30
59. Yang, W., Fedyukovich, G., Gupta, A.: Lemma synthesis for automating induction over algebraic data types. In: Schiex, T., de Givry, S. (eds.) CP 2019. LNCS, vol. 11802, pp. 600–617. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30048-7_35

60. Zhang, T., Sipma, H.B., Manna, Z.: Decision procedures for recursive data structures with integer constraints. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNCS (LNAI), vol. 3097, pp. 152–167. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-25984-8_9
61. Zhu, H., Magill, S., Jagannathan, S.: A data-driven CHC solver. In: PLDI, pp. 707–721. ACM (2018)