Solving Constrained Horn Clauses Modulo Algebraic Data Types and Recursive Functions

HARI GOVIND V K, University of Waterloo, Canada
SHARON SHOHAM, Tel-Aviv University, Israel
ARIE GURFINKEL, University of Waterloo, Canada

This work addresses the problem of verifying imperative programs that manipulate data structures, e.g., Rust programs. Data structures are usually modeled by Algebraic Data Types (ADTs) in verification conditions. Inductive invariants of such programs often require recursively defined functions (RDFs) to represent abstractions of data structures. From the logic perspective, this reduces to solving Constrained Horn Clauses (CHCs) modulo both ADT and RDF. The underlying logic with RDFs is undecidable. Thus, even verifying a candidate inductive invariant is undecidable. Similarly, IC3-based algorithms for solving CHCs lose their progress guarantee: they may not find counterexamples when the program is unsafe.

We propose a novel IC3-inspired algorithm Racer for solving CHCs modulo ADT and RDF (i.e., automatically synthesizing inductive invariants, as opposed to only verifying them as is done in deductive verification). Racer ensures progress despite the undecidability of the underlying theory, and is guaranteed to terminate with a counterexample for unsafe programs. It works with a general class of RDFs over ADTs called catamorphisms. The key idea is to represent catamorphisms as both CHCs, via *relationification*, and RDFs, using novel *abstractions*. Encoding catamorphisms as CHCs allows learning inductive properties of catamorphisms, as well as preserving unsatisfiabilty of the original CHCs despite the use of RDF abstractions, whereas encoding catamorphisms as RDFs allows unfolding the recursive definition, and relying on it in solutions. Abstractions ensure that the underlying theory remains decidable. We implement our approach in Z3 and show that it works well in practice.

CCS Concepts: • **Theory of computation → Verification by model checking**.

Additional Key Words and Phrases: Formal verification, Algebraic Data Types, Recursive Functions, Model Checking

ACM Reference Format:
Hari Govind V K, Sharon Shoham, and Arie Gurfinkel. 2022. Solving Constrained Horn Clauses Modulo Algebraic Data Types and Recursive Functions. *Proc. ACM Program. Lang.* 6, POPL, Article 60 (January 2022), 29 pages. https://doi.org/10.1145/3498722

1 INTRODUCTION

Recursive data structures, such as Lists and Trees are ubiquitous in programming. Proofs of correctness of programs that use such data structures rely on the theory of Algebraic Data Types (ADT) – the logic counter-part to data structures, and on Recursively Defined Functions (RDF), to represent properties (or abstractions) of data types. For example, length of a List, height of a Tree, are captured by RDFs.

Authors' addresses: Hari Govind V K, Department of Electrical and Computer Engineering, University of Waterloo, Canada, hgvk94@gmail.com; Sharon Shoham, Tel-Aviv University, Israel, sharon.shoham@gmail.com; Arie Gurfinkel, Department of Electrical and Computer Engineering, University of Waterloo, Canada, arie.gurfinkel@uwaterloo.ca.

2475-1421/2022/1-ART60
https://doi.org/10.1145/3498722

```
datatype List = Nil | Insert(head: int, tail: List)
  method main(y: List, i: int)
  requires length(y) == i;
  { _y, _i := y, i;
    while (*)
      invariant _i == length(_y)
       { if (_y != Nil)
         { _y, _i := _y.tail, _i - 1; }}
    assert (_i >= 0); }
```

```
function length(xs: List): int {
  match xs
  case Nil => 0
  case Insert(h, t) =>
                1 + length(t)
}
```

Fig. 1. A Dafny program that manipulates a list.

As an example, consider the imperative Dafny [Leino 2010] program in Fig. 1. The program manipulates a list represented by the variable `y`. Note that both the specification (in `requires`) and an inductive invariant (in `invariant`) require an RDF `length` (also shown in the figure). In this case, the RDF `length` appears both in the specification and the invariant, but that need not be the case. Similarly, programs without RDFs might require RDFs in their invariants (for example, via ghost code). Since the satisfiability of the combination of ADT and RDF is undecidable, even validating a candidate inductive invariant is difficult. While Dafny [Leino 2010], and similar deductive verification tools (e.g., Why3 [Filliâtre and Paskevich 2013], Viper [Müller et al. 2016]), provide ample support for reasoning with ADTs and RDFs, the situation is much worse in automated verification, where inductive invariants need to be synthesized and not just verified.

A prominent approach to automated verification, is to encode the verification condition of a given program as Constrained Horn Clauses (CHCs), such that the *solutions* to the CHCs correspond to the inductive invariants, and solve the CHCs using an off-the-shelf CHC-solver. Several such solvers are available (e.g., [Champion et al. 2018; Fedyukovich et al. 2020; Hojjat and Rümmer 2018; Komuravelli et al. 2016]) and there is an annual competition [CHC-COMP 2021]. In the case of programs with data structures, such as the one in Fig. 1, the constraints of the CHCs are over a combination of theories including Linear Integer Arithmetic (LIA) (to model `i`), ADTs (to model `y`), and RDFs (to model `length`). Supporting the combination of such theories in a CHC-solver is complicated because of the undecidability of even the most basic decision problems in this case.

In fact, several existing solvers, such as Spacer in Z3, already have some support for CHC modulo ADTs [Bjørner and Janota 2015]. However, as we show above, reasoning about ADTs almost always requires reasoning with RDFs as well, and RDFs are not supported. It is reasonable to view the relationship of RDFs to ADTs as the relationship between quantifiers and arrays – reasoning about array manipulating programs requires quantifiers for the same reasons as RDFs are needed for ADT-manipulating programs.

The key to our approach is a dual treatment of RDFs in CHCs C. On one hand, we approximate an axiom defining an RDF in C by CHCs without RDFs, through a process called *relationification*, resulting in an *equisatisfiable* CHC C_F. That is, C_F preserves both proofs (satisfiability) and counterexamples (unsatisfiability). However, it does not preserve solutions — expressing a solution for C_F might require a richer logic than the corresponding solution for C, since the former not only needs to capture an inductive invariant for the programs, but also needs to summarize the relationified RDF. On the other hand, we approximate RDFs in C by two novel abstractions that inject into C a bounded unfolding of the RDF: both abstractions apply a finite unfolding of the recursive definition, after which the *k-instantiation* abstraction treats remaining RDF applications as uninterpreted functions, while the *uninterpreted function* abstraction omits them. In both abstractions, RDFs are replaced by their partial definition — thus, gaining decidability of basic queries. Relationification enables CHC solvers to deduce inductive summaries of RDFs whereas the abstractions provide

unfoldings of RDFs. Inductive summaries and unfoldings of RDFs allow us to solve RDF constraints without facing the undecidability issues introduced by RDFs.

The combination of relationification with each of the abstractions (k-instantiation or uninterpreted functions), parameterized by an unfolding bound k, results in two systems of abstract CHCs. Each system is an over-approximation (because of the abstraction). Nonetheless, relationification ensures that the abstract CHCs are *equisatisfiable* to the original ones. The k-instantiation abstraction is perfect for finding solutions, with the number of solutions preserved by the abstraction determined by the value of k. The uninterpreted function abstraction is a CHC over ADTs only and is perfect for showing unsatisfiability (i.e., finding counterexamples). The abstractions, parameterized by k, thus, create a hierarchy of abstractions (shown in Fig. 4).

We present an algorithm, Racer, that operates in the space of the hierarchy of k-instantiation abstractions, dynamically adjusting k, while simultaneously searching through the counterexamples in the uninterpreted function abstraction. While this sounds complex, it fits well into an IC3-style framework that iteratively explores bounded unfoldings of the system. Remarkably, this integration seamlessly combines learning inductive properties of RDFs with learning inductive invariants of the system, while exploiting the SMT support for non-inductive RDF reasoning via incremental unfolding of RDFs.

An important characteristic of our algorithm is that it is guaranteed to make progress and to discover counterexamples, if they exist. The problem of CHC solving is more general than SMT-solving (i.e., inferring invariants vs. checking invariants). Even CHC modulo LIA is undecidable, thus, there is little hope for completeness results when considering CHC solving. On the other hand, unsatisfiability of CHCs (and FOL, in general) is recursively enumerable. For CHCs, unsatisfiability corresponds to the existence of a counterexample to safety. As a result, progress — which implies, in particular, discovery of counterexamples — is key.

The integration of relationification with abstractions of RDFs is crucial for ensuring progress while preserving solutions of the CHCs. While k-deep unfolding alone does not suffice for making progress, relationification ensures progress as it allows to learn bounded properties of RDFs. On the other hand, relationification alone does not preserve solutions of the CHCs due to the inability to express summaries of RDFs. Thus, the combination of relationification with unfolding of RDFs is critical for maintaining progress while preserving (many) solutions. It allows to combine inference of inductive summaries of RDFs, which are needed to establish validity of solutions, with some unfoldings, guided by the property.

We develop a single procedure for solving CHCs with ADTs, RDFs, and LIA, while maintaining progress, and without affecting performance on LIA. This is in contrast to other related work, which either automates induction assuming that arithmetic is axiomatized [Reynolds and Kuncak 2015; Rosén and Smallbone 2015] or does not admit RDFs [Fedyukovich et al. 2020; Hojjat and Rümmer 2018; Komuravelli et al. 2016].

In our exposition, we restrict RDFs to catamorphisms (a.k.a. generalized folds). The RDF `length` in Fig. 1 is an example. Extending our ideas to other well defined RDFs is explained in Sec. 5.

Limitations. Our current approach is limited in two ways. First, we assume that the RDFs are specified. That is, we only synthesize inductive invariants, but not the supporting functions. Second, we do not synthesize new applications of RDFs, relying on a form of term abstraction [Alberti et al. 2012]. We leave addressing these orthogonal problems to future work.

Contributions. The paper makes the following contributions: (1) a new hierarchy of abstractions for CHC modulo ADT and RDF that combine relationification with bounded unfoldings of RDFs, thus, preserving counterexamples and (many) solutions; (2) an IC3-based procedure Racer for solving CHC modulo ADT and RDF that searches through the hierarchy of abstractions; It integrates

into and extends the Spacer framework, and provides progress guarantees; (3) an implementation of Racer in Z3 that works with a combination of ADT, RDF, and other theories, such as LIA. (4) an empirical evaluation on a variety of benchmarks.

2 OVERVIEW

In this section we give an overview of our approach using a motivating example. Our focus is automatic safety verification of imperative programs that manipulate Algebraic Data Types (ADTs), such as lists, trees, or queues. Such programs and their specifications often require Recursively Defined Functions (RDFs), such as *length*, *size*, etc. Thus, we must support reasoning about the combination of ADTs and RDFs.

The key challenge of automated safety verification is automatically synthesizing and validating inductive invariants that summarize the behavior of all the loops. In case of ADTs, expressing invariants often requires RDFs, and, sometimes, RDFs that are not even present in the original program. For example, inductive invariants of list-manipulating programs often require to use the length of a list (and other recursive functions that compute inductive properties of lists). Thus, automatic reasoning over ADTs requires (a) synthesis of inductive invariants over ADTs and RDFs, and (b) synthesis of useful RDFs. In this paper, we focus on the first challenge – invariant synthesis. We assume that RDFs and their applications to program variables are either heuristically extracted from the program or are provided by the users. Synthesizing useful RDFs can be done as an abstraction-refinement loop over our work.

2.1 Motivating Example

To illustrate our approach, consider the list manipulating program in Fig. 1. `List` is a list ADT with two constructors `Nil` and `Insert`. `length` is an RDF that computes the length of a list. The method `main` takes as input a list `y` and its length `i`. It removes elements from the list one by one in a loop, decreasing `i` after each removal. Whenever the loop exits, it is asserted that the variable `i` has a non negative value. The program is safe, as witnessed by the following inductive invariant `i == length(y)`. While this program is simple, automatically verifying its safety is difficult for many existing verification techniques. Specifically, there are multiple theories involved that require to simultaneously reason about integer variables (i.e., variable `i`), ADT variables (i.e., variable `y`), and RDFs (i.e., function `length`). Furthermore, the proof that the suggested invariant is inductive requires guessing an inductive property of `length`: `length(y)` is non-negative for any list `y`.

An effective technique for automatic program verification is reducing verification to satisfiability of Constrained Horn Clauses (CHCs). A CHC encoding of our example program from Fig. 1 is shown in Fig. 2a. In this encoding, the uninterpreted predicate *Inv* summarizes all behaviours of the `main` method. Essentially, it represents a desired inductive invariant of the loop. *Inv* takes three arguments: the first two are the same as those of `main`, and the third encodes the length of the list `y`. The third argument is required so that `length(y)` can appear in an inductive invariant.

A program is safe with respect to a specification if its CHC encoding is satisfiable (as a First Order Logic formula). A *solution* to CHCs is a mapping from all of the uninterpreted predicates to FOL formulas (in some theory), such that replacing predicates by the corresponding formulas makes the resulting FOL formula valid. For instance, in the CHC encoding of our example (Fig. 2a), a solution assigns to predicate $Inv(y, i, j)$, the formula $(i = j)$, where j encodes the length of list `y`, as captured by the constraint $length(y) = j$. Hence, this is essentially the inductive invariant `i == length(y)`. Representing the RDF application $length(y)$ by j follows the idea of *term abstraction*. It allows us to separate the task of synthesizing the required RDF applications from the task of inferring inductive invariants over them, which is the focus of the paper. Namely, instead of synthesizing RDF applications in the process of inferring an invariant (solution), we infer invariants over variables

$$length(y) = i \Rightarrow Inv(y, i, i)$$
$$Inv(y, i, j) \wedge length(y) = j \wedge y \neq nil \wedge y' = tail(y) \wedge length(y') = j' \Rightarrow Inv(y', i - 1, j')$$
$$Inv(y, i, j) \wedge length(y) = j \wedge i < 0 \Rightarrow \bot$$

(a) CHC modulo ADT and RDF encoding safety of the program from Fig. 1.

$$y = nil \wedge z = 0 \Rightarrow Length(y, z)$$
$$y \neq nil \wedge Length(tail(y), l) \wedge z = 1 + l \Rightarrow Length(y, z)$$
$$Length(y, i) \Rightarrow Inv(y, i, i)$$
$$Inv(y, i, j) \wedge Length(y, j) \wedge y \neq nil \wedge y' = tail(y) \wedge Length(y', j') \Rightarrow Inv(y', i - 1, j')$$
$$Inv(y, i, j) \wedge Length(y, j) \wedge i < 0 \Rightarrow \bot$$

(b) CHCs C_F obtained by relationifying the RDFs in Fig. 2a

$$y = nil \wedge z = 0 \Rightarrow Length(y, z)$$
$$y \neq nil \wedge Length(tail(y), l) \wedge z = 1 + l \Rightarrow Length(y, z)$$
$$Length(y, i) \wedge i = length(y) \Rightarrow Inv(y, i, i)$$
$$Length(y, j) \wedge j = length(y) \wedge Length(y', j') \wedge j' = length(y') \wedge$$
$$Inv(y, i, j) \wedge y \neq nil \wedge y' = tail(y) \Rightarrow Inv(y', i - 1, j')$$
$$Length(y, j) \wedge j = length(y) \wedge Inv(y, i, j) \wedge i < 0 \Rightarrow \bot$$

(c) C_{fF}: the result of adding back the *length* RDF to relationified CHCs C_F from Fig. 2b.

Fig. 2. Example CHCs for (a) encoding of Dafny program from Fig. 1; (b) relationification, (c) relationifications with RDFs. Modifications and additions are highlighted in yellow and green, respectively.

that represent a predefined set of RDF applications. With this encoding, deciding whether the original program is safe is reduced to inferring solutions to the corresponding CHC encoding of its verification condition. We stress that inferring solutions (i.e., inductive invariants) is much harder than checking that a candidate invariant is inductive, as traditionally done in deductive verification.

Note that it is possible that a set of CHCs is satisfiable, but has no solution in a given theory. That is, the FOL model that witnesses satisfiability is not definable in the fixed language of solutions. On the flip side, a program is unsafe with respect to a specification if its CHC encoding is unsatisfiable. In this case, the program exhibits a (finite) counterexample: a feasible execution of the program that violates the specification. From perspective of CHCs, this means that there is a refutation resolution proof (i.e., a proof deriving false) from the CHCs.

2.2 Challenges for Solving CHCs Modulo RDFs

Solving CHCs is a challenging problem in general, and has received a lot of attention (e.g., [De Angelis et al. 2020; Fedyukovich et al. 2017; Grebenshchikov et al. 2012; Hoder and Bjørner 2012; Hojjat and Rümmer 2018; Komuravelli et al. 2016; Zhu et al. 2018]). CHCs modulo ADTs and RDFs bare additional challenges:

$$y = nil \wedge z = 0 \Rightarrow Length(y, z)$$
$$y \neq nil \wedge Length(tail(y), l) \wedge z = 1 + l \Rightarrow Length(y, z)$$
$$Length(y, i) \wedge i = length_{uf}(y) \wedge i = ite(y = nil, 0, 1 + length_{uf}(tail(y))) \Rightarrow Inv(y, i, i)$$
$$Length(y, j) \wedge j = length_{uf}(y) \wedge j = ite(y = nil, 0, 1 + length_{uf}(tail(y))) \wedge$$
$$Length(y', j') \wedge j' = length_{uf}(y') \wedge j' = ite(y' = nil, 0, 1 + length_{uf}(tail(y'))) \wedge$$
$$Inv(y, i, j) \wedge y \neq nil \wedge y' = tail(y) \Rightarrow Inv(y', i - 1, j')$$
$$Length(y, j) \wedge j = length_{uf}(y) \wedge j = ite(y = nil, 0, 1 + length_{uf}(tail(y))) \wedge$$
$$Inv(y, i, j) \wedge i < 0 \Rightarrow \bot$$

(a) C^1_{fF}: the 1-instantiation abstraction of CHCs C_{fF} from Fig. 2c. $length_{uf}$ is an uninterpreted function.

$$y = nil \wedge z = 0 \Rightarrow Length(y, z)$$
$$y \neq nil \wedge Length(tail(y), l) \wedge z = 1 + l \Rightarrow Length(y, z)$$
$$Length(y, i) \wedge i = ite(y = nil, 0, 1 + w) \Rightarrow Inv(y, i, i)$$
$$Length(y, j) \wedge j = ite(y = nil, 0, 1 + w) \wedge Length(y', j') \wedge j' = ite(y' = nil, 0, 1 + w') \wedge$$
$$Inv(y, i, j) \wedge y \neq nil \wedge y' = tail(y) \Rightarrow Inv(y', i - 1, j')$$
$$Length(y, j) \wedge j = ite(y = nil, 0, 1 + w) \wedge Inv(y, i, j) \wedge i < 0 \Rightarrow \bot$$

(b) $C^{k\uparrow}_{fF}$: Uninterpreted functions abstraction of CHCs C_{fF} from Fig. 2c.

Fig. 3. Examples of (a) k-instantiation and (b) uninterpreted function abstractions of CHCs from Fig. 2c.

Undecidability of validating solutions. In the presence of RDFs, determining validity of FOL formulas is undecidable. Thus, even checking whether a candidate formula is an inductive invariant (a step used by many existing verification techniques) is undecidable. Validity may depend on properties of RDFs that must be established inductively (i.e., no amount of finite unfolding of the recursive definition is sufficient). Applying induction requires coming up with an inductive hypothesis, which, intuitively, is a source of undecidablity.

The undecidability challenge significantly limits applicability of guess-and-check techniques, such as Houdini [Flanagan et al. 2001; Flanagan and Leino 2001], in this context. This extends to all guess-and-check techniques including more recent approaches based on machine learning [Garg et al. 2016] and grammar-based synthesis [Fedyukovich et al. 2020]. One exception is the CVC4 SMT solver that combines candidate enumeration with induction [Reynolds and Kuncak 2015]. However, the technique in [Reynolds and Kuncak 2015] is not fully automated (but rather user or problem guided). It assumes that required inductive hypotheses are either part of the input, or are provided by the user as sub-goals. This is justified by their application domain (automating deductive verification), but is not very effective in an automated verification setting, as recently shown in [De Angelis et al. 2020]. There are also known sub-classes of RDFs for which validity is decidable. One such class of *infinitely surjective catamorphisms (ISCs)* over ADTs is presented by Suter et al. [Suter et al. 2010]. However, it is not clear that determining that an RDF is of an appropriate class (i.e., an ISC) is decidable.

Incompleteness of RDF elimination. As RDFs make validating solutions undecidable, a natural direction of attack is to reduce the problem by somehow eliminating RDFs, while preserving the key characteristics of the verification task. One approach is *relationification*: encoding RDFs as

CHC predicates, constrained by additional Horn clauses. For example, relationification of the CHCs in Fig. 2a gives us the CHCs in Fig. 2b. Relationified CHCs do not contain RDFs. Instead, they are over ADTs (and other background theories). Therefore, checking the validity of solutions is decidable. Intuitively, relationification replaces recursive functions by their implementation. One of our results is to show that while relationification preserves satisfiability, it loses solutions over the language of CHCs. Roughly, this is because a solution to the CHCs should now also include a summary of the relationified RDF. However, there are cases where such a summary cannot be specified without RDFs (see Ex. 5.8).

2.3 Our Approach

In this paper, we present a new approach for solving CHCs modulo ADT and RDF that addresses the above challenges. The core of our approach is the *combination* of relationified predicates with a novel *abstraction* of RDFs based on finite RDF unfoldings.

The use of (abstraction of) RDFs allows to express solutions that cannot be expressed without RDFs, thus, tackling the second challenge. Abstractions are key to ensuring that checking validity of solutions remains decidable (tackling the first challenge). Unfortunately, abstractions may not be sufficient to validate solutions, specifically, in the cases where finite unfoldings do not suffice. This is where the interplay with the relationified predicates, that are conjoined with the RDF abstractions, comes into play: relationified predicates allow to use Property Directed Reachability (PDR) to automatically generate necessary inductive hypotheses and lemmas that are needed in order to validate solutions. However, this is not the only role of the relationified predicates. The Horn clauses that encode the relationified predicates ensure that despite the RDF abstraction, unsatisfiability is preserved. Namely, if the original CHC is unsatisfiable, so is the transformed one. Indeed, a key characteristic of our approach is that it is *refutationally complete* — whenever the (original) CHCs are unsatisfiable (i.e., the program is unsafe), our approach is guaranteed to terminate with a finite counterexample. In fact, our approach has a stronger property from which the former follows: the approach is ensured to make progress in verifying *bounded safety* for increasing bounds.

Relationification side by side with RDFs. The first step in our approach is to extend a given set of CHCs modulo RDFs with relationification of the RDFs. That is, we conjoin the CHCs with clauses that constrain the relationified predicates and conjoin each RDF application with an application of the relationified predicate. For example, the result of this transformation on CHCs in Fig. 2a is shown in Fig. 2c. The newly added clauses, and newly added predicate applications are highlighted in green and yellow, respectively. The resulting CHCs preserve both solutions of the original CHCs (when they are satisfiabile), and unsatisfiability (when the original CHCs are unsatisfiable).

Abstractions of RDFs. The second step in our approach is a new IC3-style algorithm, called Racer, that employs two orthogonal abstractions simultaneously in order to solve the augmented CHCs. These abstractions, called *k-instantiation* and *uninterpreted functions* abstraction, respectively, are key to avoiding the undecidability barrier (for solution validation).

The k-instantiation abstraction unfolds all occurrences of RDFs in a formula k times and replaces any remaining RDFs with uninterpreted functions (UF). Applying the k-instantiation abstraction to CHCs modulo ADTs and RDFs, gives us a set of abstract CHCs modulo ADTs and UF.[1] The result of applying the 1-instantiation abstraction to the CHCs in Fig. 2c is shown in Fig. 3a. We show that, similar to the other transformations, the k-instantiation abstraction of CHCs is equi-satisfiable to the original CHCs, but, more importantly, it has more solutions than relationified CHCs. In fact, there is a hierarchy: all solutions to $(k-1)$-instantiation abstraction are also solutions to

[1]Technically, the result is not in FOL because UFs are implicitly universally quantified. See Sec. 6.2 for a precise definition.

k-instantiation abstraction (Fig. 4), and all of them are solutions to the original CHCs. Furthermore, validating solutions of the abstraction of the CHCs is decidable, as it only involves ADTs and UF, but no RDFs. Hence, the k-instantiation abstraction is particularly useful for finding solutions. Note that while the abstraction makes validating solutions decidable, relationified predicates provide the supporting inductive lemmas (about RDFs) that are often necessary for solutions to be validated.

The uninterpreted function abstraction removes all occurrences of UF from a given formula (a detailed definition is given in Sec. 6.1). Thus, it results in even simpler CHCs — without UFs. An example of the uninterpreted functions abstraction of the CHCs in Fig. 3a is shown in Fig. 3b. The uninterpreted function abstraction also preserves (un)satisfiability, thanks to the relationified predicates. This abstraction is useful for proving unsatisfiability of the original CHCs, as refutational completeness of first order logic ensures the existence of a (finite) refutation proof for the abstraction.

While the description above might suggest that we explicitly construct the abstractions, this is not the case. The abstractions are tightly integrated as part of an IC3-style algorithm, Racer (Alg. 1 in Sec. 7). Racer uses k-instantiation abstraction to block infeasible counterexamples and to infer useful inductive lemmas over the predicates (including the predicates that define RDFs), and uninterpreted function abstraction to extend partial counterexamples (until they are either shown to be infeasible or become complete). The value for k, in both abstractions, is adjusted dynamically by Racer. While Racer does not guarantee to terminate — the underlying problem, after all, is undecidable, it does guarantee to find a counterexample if one exists.

For our running example, Racer produces the following solution:

$$Inv(y, i, j) \triangleq i = j \qquad\qquad Length(y, j) \triangleq j \geq 0$$

Note that the solution for *Length* is an inductive property of the `length` RDF.

Outline. The rest of the paper is structured as follows. We review the necessary background in Sec. 3, and define CHC modulo ADTs and RDFs in Sec. 4. Reduction from CHC modulo RDF to CHC is shown in Sec. 5, followed by the definition of RDF abstractions for CHCs in Sec. 6. The main algorithm is presented in Sec. 7 and evaluated in Sec. 8. Related work is discussed in Sec. 9 and Sec. 9 concludes the paper.

3 BACKGROUND

In this section, we provide a brief background on first order logic, satisfiability modulo theories, the theory of ADTs, which this paper focuses on, and Constrained Horn Clauses.

3.1 Satisfiability Modulo Theories

We work in standard many-sorted First Order Logic with equality. Terms and formulas are defined over a signature Σ which consists of a set of sorts, a set of predicate symbols and a set of function symbols, each with a designated non-negative arity and sorts. We refer to zero-ary function symbols as constants. A term t is either a (sorted) variable x, or a function symbol applied to terms, $f(t_1, \ldots, t_n)$, respecting the arity and sorts of f. An atomic formula is an equality applied to terms of the same sort, $t_1 = t_2$, or a predicate symbol applied to terms, $p(t_1, \ldots, t_n)$, respecting the arity and sorts of p. A formula is a Boolean combination of atomic formulas, possibly with quantification. We use the special term $ite(c, t_1, t_2)$, where c is a formula, and t_1, t_2 are terms of the same sort, to denote an "if then else" operation. A model M consists of a nonempty domain for each sort, and a valuation function that maps each function and predicate symbol to its interpretation. For a closed formula φ, $M \models \varphi$ is defined in the usual way.

We assume an underlying theory $\mathcal{T}$ that interprets some sorts, function and predicate symbols. A symbol interpreted by the theory is called an interpreted symbol. We refer to interpreted constants

as values. Theory $\mathcal{T}$ may either be defined by a set of axioms (in which case the theory consists of the deductive closure of the axioms), or by a set of intended models (in which case the theory consists of all formulas that are true in these models). We consider *satisfiability modulo theories*, and say that a formula φ is *satisfiable* modulo theory $\mathcal{T}$ if there is a *model* M of $\mathcal{T}$ that satisfies the formula, denoted $M \models_{\mathcal{T}} \varphi$. Otherwise, the formula is *unsatisfiable*. φ is *valid* modulo $\mathcal{T}$ if every model of $\mathcal{T}$ satisfies φ. We say that two formulas, θ_1 and θ_2, are *equisatisfiable*, denoted $\theta_1 \approx \theta_2$, if it holds that there exists $M \models_{\mathcal{T}} \theta_1$ if and only if there exists $M' \models_{\mathcal{T}} \theta_2$. We omit $\mathcal{T}$ and write $M \models \varphi$ when the theory is clear from context.

By convention, we use non bold symbols (like f) to refer to syntactic entities and bold symbols (like $\boldsymbol{f}$) to refer to their semantics i.e., their valuation in a model. Given a model M over a signature Σ that may or may not include a, we use the notation $M[a \mapsto \boldsymbol{a}]$ to mean the model over $\Sigma \cup \{a\}$ obtained from M when a is interpreted to $\boldsymbol{a}$.

Literals and clauses are defined as usual: a literal is an atomic formula or its negation, a clause is a disjunction of literals. Throughout the paper, we say that a formula θ is *pure* with respect to a function symbol f, if f only appears in positive literals of the form $f(\bar{t}) = x$ where f does not appear in $\bar{t}$, and x is a variable. We call such formulas f-pure to emphasize f. Note that every quantifier-free formula θ can be converted into an equivalent f-pure formula by introducing existentially quantified variables that represent the f applications. For example, $p(f(f(x)))$ may be rewritten into $\exists y, z \cdot f(x) = y \wedge f(y) = z \wedge p(z)$.

3.2 Theory of Algebraic Data Types

The theory of Algebraic Data Types (ADT) is defined over a signature Σ containing a sort τ for an algebraic datatype (for simplicity, we assume only one such sort). The signature Σ includes function symbols for *constructors* and *selectors*, and predicate symbols for *testers*. A constructor for sort τ, denoted C, returns an element of sort τ (its arguments may be of any sort, including τ). A selector for the ith argument of C, denoted S_C^i, receives an element of sort τ and returns an element of the sort of the ith argument of C. A tester for C, denoted IS_C, is defined over sort τ. The theory of ADTs is defined by an *initial model* of the following axiom schema, ξ:

$$\forall \bar{x} \cdot IS_C(C(\bar{x})) \qquad \forall \bar{x} \cdot \neg IS_{C'}(C(\bar{x})) \qquad \forall \bar{x} \cdot S_C^i(C(\bar{x})) = x_i$$

where $\bar{x} = \{x_1, \ldots, x_n\}$ and $C \neq C'$. An initial model is a model where the domain of sort τ consists of all *constructor terms*: these are ground terms where the only ADT function applications are of constructors. The theory of ADT consists of the set of formulas that hold in an initial model of ξ. The theory of ADTs is decidable (even with quantifiers) [Zhang et al. 2004].

Note that an initial model cannot be specified in First Order Logic. However, many SMT solvers support reasoning over it. Accordingly, in the sequel, all models we consider are isomorphic to an initial model. In particular, we write $M \models \theta$ to mean that M is isomorphic to an initial model that satisfies θ.

The List ADT. For convenience of presentation, throughout the paper, we use a combination of the List ADT and Linear Integer Arithmetic, i.e., Lists of Integers, denoting them simply as Lists. We stress that our approach is more general. It applies to other ADTs, including mutually recursive ones. The List datatype is defined using a signature that consists of sort s for the elements stored in lists, which is in our case sort Int, and sort List_s for lists. It has two constructors:

- *nil* is a nullary constructor for constructing an empty list, and
- *insert* receives an element of sort s and a list and constructs a list where the element is added to the head of the list.

The *insert* constructor is associated with two selectors: head for the first argument, and tail for the second argument.

3.3 Constrained Horn Clauses

Given a signature Σ, a background theory $\mathcal{T}$ over Σ, and a set P of uninterpreted predicate symbols not in Σ (but defined over the sorts in Σ), a Constrained Horn Clause (CHC) is a closed formula of the form

$$\forall \bar{x}, \bar{x}' \cdot \phi(\bar{x}, \bar{x}') \wedge \bigwedge_i p_i(\bar{x}_{p_i}) \Rightarrow h(\bar{x}'_h) \tag{1}$$

where $p_i \in P$, $h \in P \cup \{\bot\}$, $\bar{x}' = \{x' \mid x \in \bar{x}\}$, ϕ is a quantifier-free formula over Σ with free variables in $\bar{x} \cup \bar{x}'$, $\bar{x}_{p_i} \subseteq \bar{x}$, and $\bar{x}'_h \subseteq \bar{x}'$. The same unknown predicate $p \in P$ can appear multiple times in a CHC with different arguments (i.e., in the equation above p_i is not necessary distinct from p_j when $i \neq j$). The left hand side of the implication is called the *body* or *tail* of the CHC, and the right hand side is called its *head*. We refer to ϕ as the *constraint*. A set of CHCs is a conjunction of CHCs. With abuse of notation, we sometimes refer to a set of CHCs as a CHC. A (single) CHC is *linear* if the tail has at most one uninterpreted predicate symbol. A set of CHCs is linear if all the CHCs in it are linear. Otherwise, the set of CHCs is *non linear*. When a set of CHCs is satisfiable (modulo $\mathcal{T}$), a *solution* is a model, including an interpretation of all the predicates in P, that satisfies all the CHCs. Typically, we are interested in solutions that are expressed as formulas over some class of formulas $\mathcal{A}$ [Bjørner et al. 2015]:

Definition 3.1 ($\mathcal{A}$-Solutions). Let $\mathcal{A}$ be a class of formulas. An $\mathcal{A}$-solution for a set of CHCs C assigns to each predicate symbol $p \in P$ a formula $\theta_p(\bar{y}) \in \mathcal{A}$ whose free variables $\bar{y}$ correspond to the arguments of p such that replacing each predicate application $p(\bar{x}_p)$ in C by $\theta_p[\bar{y} \mapsto \bar{x}_p]$ results in a valid closed formula (modulo the background theory).

It is possible that a set of CHCs is satisfiable, but has no $\mathcal{A}$-solution.

Programs as CHCs. A set of CHCs can encode program semantics. A non-recursive program is encoded using linear CHCs. All linear CHCs are reducible to a set of CHCs with a single unknown predicate *Inv* and exactly 3 clauses (free variables are implicitly universally quantified):

$$Init(\bar{x}) \Rightarrow Inv(\bar{x}) \qquad Inv(\bar{x}) \wedge Tr(\bar{x}, \bar{x}') \Rightarrow Inv(\bar{x}') \qquad Inv(\bar{x}) \wedge Bad(\bar{x}) \Rightarrow \bot$$

In the context of safety verification, assignments to $\bar{x}$ are called *states*, *Init* represents the set of initial states of the program, *Tr* represents its set of transitions and *Bad* represents a set of bad states. In this case, a solution for *Inv* is an *inductive invariant* that proves that the program is *safe*: no bad state is reachable from an initial state via the program's transitions. When the CHC is unsatisfiable, there exists a *counterexample* to safety. Counterexamples to the safety property are derived from resolution proofs of unsatisfiability of CHCs. Recursive programs (and their safety) are encoded as non linear CHCs. In this case, a solution to the CHCs contain necessary function summaries to prove that the program is *safe*. Similarly, counterexamples to safety are derived from resolution proofs of unsatisfiability.

4 CHCS MODULO ADT AND CATAMORPHISMS

In this work, we are interested in verifying safety of programs specified using the theory of ADT and whose correctness arguments are specified using recursively defined functions. Specifically, we are interested in *catamorphisms*. In this section, we define catamorphisms (Sec. 4.1), and explain the conventions we use when considering CHCs that encode safety of programs (Sec. 4.2).

4.1 Catamorphisms

Recursive functions are in general not expressible in First Order Logic. In SMT they can be specified by *function axioms*, which can precisely capture the definition of the recursion, but relax the least-fixpoint semantics (e.g., [Löding et al. 2018]). A function axiom is a formula of the form $\forall \bar{x} \cdot f(\bar{x}) = t(\bar{x})$ where t is a term of an appropriate sort. t can contain f itself as well as other function (or constant) symbols. We call a function symbol f for which a function axiom is given, a recursively defined function (RDF), to distinguish it from other uninterpreted function symbols. However, there is really no easy way to tell when a function axiom *defines* a function. Some function axioms are unsatisfiable, for example, $\forall x \cdot f(x) = \neg f(x)$. Some axioms provide a complete definition of a function, e.g., $\forall x \cdot f(x) = ite(x \leq 0, 0, x \times f(x-1))$, which has a unique interpretation over the standard model of arithmetic, and some do not, e.g., $\forall x \cdot f(x) = f(x+1) - 1$, which is satisfied by the interpretation $\boldsymbol{f}(x) = x$, but also by $\boldsymbol{f}(x) = x + 1$, $\boldsymbol{f}(x) = x + 2$, etc. While SMT-LIB does not restrict function axioms to uniquely define a function, our work addresses RDFs that are uniquely defined by their axioms. To avoid the need to characterize such functions, we focus on the special case of catamorphisms. In Sec. 5 we discuss how to extend our results to other well defined functions.

Catamorphisms are a notable example of function axioms that specify well defined functions over ADTs. A catamorphism (or, generalized fold) is an RDF that abstracts the content of an ADT into a *single* value. Common abstractions, such as length of a list, height of a tree, set of all elements of a tree, are all examples of catamorphisms. Given an interpreted sort r for the range of the abstraction, a catamorphism from the list ADT List_s to r is specified using a base case value, b, of sort r, and a term, denoted $y \oplus z$, of sort r, where y and z are variables of sorts s and r, respectively, and all symbols are interpreted. Intuitively, the base case value b is used to define the value of the catamorphism for nil, while $y \oplus z$ is used to define the value in the recursive case of a list x created by the *insert* constructor, in which case the value is defined by applying $\oplus$ on the element $y = head(x)$ inserted to the list and on the value z recursively defined for $tail(x)$. Formally, given $b, \oplus$, the following function axiom defines f as a catamorphism over List_s:

$$\forall x \cdot f(x) = ite(x = nil, b, head(x) \oplus f(tail(x)))$$

We denote such a function axiom $\varphi_{(\!|b,\oplus|\!)}$ and call it a catamorphism axiom for f.

Example 4.1. The *length* of a list is a catamorphism from List to Int specified by:

$$\varphi_{(\!|0,+1|\!)} \triangleq \forall x \cdot length(x) = ite(x = nil, 0, 1 + length(tail(x)))$$

The theory of ADTs with catamorphisms (and, more generally, RDFs) is undecidable.

4.2 CHCs Modulo ADTs and Catamorphisms

In this section, we present the form of CHCs we are interested in. We also list out a number of assumptions so that the presentation in later parts of the paper is easier to follow. We encode programs that manipulate datatypes as CHCs modulo a theory $\mathcal{T}$ that includes the theory of ADTs. The encoding further uses catamorphisms, specified via function axioms. In the sequel, we fix a signature Σ that consists of the sorts s, r, and List_s, and constructors, selectors, and testers for lists. The signature Σ also includes a function symbol f of sort $\text{List}_\text{s} \mapsto \text{r}$ that is used to specify a catamorphism. We denote by $\varphi_{(\!|b,\oplus|\!)}$ the catamorphism axiom for f.

In the paper, we consider satisfiability of $C_f \wedge \varphi_{(\!|b,\oplus|\!)}$, where C_f is a set of CHCs of the form:

$$\begin{aligned} Init(\bar{y},\bar{z}) \wedge \big(\bigwedge_i f(y_i) = z_i\big) &\Rightarrow Inv(\bar{y}',\bar{z}') \\ Inv(\bar{y},\bar{z}) \wedge Tr(\bar{y},\bar{z},\bar{y}',\bar{z}') \wedge \big(\bigwedge_i f(y_i) = z_i \wedge f(y_i') = z_i'\big) &\Rightarrow Inv(\bar{y}',\bar{z}') \\ Inv(\bar{y},\bar{z}) \wedge \big(\bigwedge_i f(y_i) = z_i\big) \wedge Bad(\bar{y},\bar{z}) &\Rightarrow \bot \end{aligned} \tag{2}$$

where $y_i \in \bar{y}$, $z_i \in \bar{z}$, y_i is of sort List_s and z_i is of sort r. Repetitions are allowed: y_i can be the same as y_j. We further assume that $Init$, Tr, and Bad do not contain any f applications, hence the bodies of the CHCs are f-pure. Such a form may be obtained by purification. $Inv \notin \Sigma$ is the only unknown predicate. Throughout the paper, we use C_f to refer to CHCs of the form Eq. (2), and we refer to $C_f \wedge \varphi_{(\!|b,\oplus|\!)}$ as CHCs modulo function axiom.

CHCs like those in Eq. (2) can be directly constructed from functional programs and imperative programs.

The assumptions about the sort of y_i, z_i, f, that there is only one ADT, and one f are for presentation purposes only. We stress that our ideas and our implementation work for multiple catamorphisms defined over any ADT, mutually recursive ADTs, mutually recursive catamorphisms, and non-linear CHCs. Relaxing these assumptions does not require any new algorithmic insights, and, in the implementation required no change from the underlying CHC solver.

Solutions. In this paper, we limit our attention to $\mathcal{A}$-solutions of C_f modulo $\varphi_{(\!|b,\oplus|\!)}$, where $\mathcal{A}$ is the class of quantifier-free formulas over the signature $\Sigma \setminus \{f\}$. That is, f is not allowed to appear in the solution. We leave lifting this assumption to future work. Solutions that involve f can be accounted for, even with the restricted language, if the CHCs are constructed in such a way that the constraints in the clauses contain all the necessary f applications to prove safety (i.e., to obtain a solution), and that these applications are captured by variables that are included as arguments to solutions. In particular, term abstraction [Alberti et al. 2012] may be used to introduce auxiliary variables that represent f-applications that are needed in order to express the solution, and augment the bodies of the CHCs with them. For example, in Fig. 2, the variable j that is added as an argument to Inv is an auxiliary variable introduced to express $length(y)$.

Motivation for our work. Existing solvers for CHCs make multiple satisfiability queries over the background theory. Unfortunately, satisfiability becomes undecidable when considering the theory of ADTs with RDFs. The remainder of the paper is dedicated to showing how existing solvers may be extended to handle such CHCs while ensuring that all satisfiability queries remain decidable.

5 REDUCING CHCS MODULO CATAMORPHISMS TO CHCS

To alleviate the problems arising from undecidability of catamorphisms, we show how to reduce CHCs modulo catamorphisms to pure CHCs whose underlying theory is decidable. Specifically, we show how to reduce $C_f \wedge \varphi_{(\!|b,\oplus|\!)}$, a set of CHCs with an axiomatized catamorphism f, to an equisatisfiable set of CHCs C_F without f. The key idea is to encode f by a fresh predicate (relation) symbol F, of appropriate sort, i.e., if f is of sort $\text{List}_\text{s} \mapsto \text{r}$, then F is a predicate over sorts $\text{List}_\text{s} \times \text{r}$. The predicate F is added to the set of unknown CHC predicates (which in our case consists of Inv only), and is used instead of f in C_f. The challenge is to express the axiom for f ($\varphi_{(\!|b,\oplus|\!)}$) as a CHC over F, while preserving equisatisfiability.

From CHCs with f to CHCs with F. We first address the CHCs C_f. As the bodies of these CHCs are f-pure, we can use a simple substitution to replace f by F.

Definition 5.1 (Relationification). Let C_f be a set of CHCs whose bodies are f-pure. We denote by $C_f[f \mapsto F]$ the set of CHCs obtained by replacing all equalities of the form $f(t) = z$ in the constraints of C_f, with $F(t, z)$.

The CHCs resulting from relationification do not contain any occurrences of the function symbol f. On the other hand, they contain the new unknown predicate F. For example, relationification of the CHC $\forall x, y, z \cdot A(x) \wedge f(x) = y \wedge f(y) = z \Rightarrow B(z)$ gives us $\forall x, y, z \cdot A(x) \wedge F(x, y) \wedge F(y, z) \Rightarrow B(z)$.

The following lemma outlines the conditions in which relationification preserves satisfiability:

Lemma 5.2. *Let M be a model, and $\boldsymbol{f}$ and $\boldsymbol{F}$ be interpretations of f and F over the universe of M, respectively.*

- *If $M[f \mapsto \boldsymbol{f}] \models C_f$ and $\boldsymbol{F}$ is the graph[2] of $\boldsymbol{f}$ then $M[F \mapsto \boldsymbol{F}] \models C_f[f \mapsto F]$.*
- *If $M[F \mapsto \boldsymbol{F}] \models C_f[f \mapsto F]$ and $\boldsymbol{F}$ is an over approximation of the graph of $\boldsymbol{f}$ then $M[f \mapsto \boldsymbol{f}] \models C_f$.*

Proof. The first statement is straight forward. For the second statement, we have $M[F \mapsto \boldsymbol{F}] \models C_f[f \mapsto F]$. Let $\boldsymbol{G_f}$ be a stronger relation than $\boldsymbol{F}$. Since F occurs only on the tail of clauses in $C_f[f \mapsto F]$, we have that $M[F \mapsto \boldsymbol{G_f}] \models C_f[f \mapsto F]$. In particular, $\boldsymbol{G_f}$ can be the relation $\{(\boldsymbol{x}, \boldsymbol{z}) \mid \boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{z}\}$, i.e., the graph of $\boldsymbol{f}$. This implies that $M[f \mapsto \boldsymbol{f}] \models C_f$. □

From function axioms to CHCs. Now, we move on to constructing CHCs to capture the catamorphism axiom.

Definition 5.3 (Positive relation constraint). Given a catamorphism axiom $\varphi_{(\!|b,\oplus|\!)}$ for f and a fresh relation symbol F, the positive relation constraint for $\varphi_{(\!|b,\oplus|\!)}$ is

$$RC^+_F(\varphi_{(\!|b,\oplus|\!)}) = \begin{pmatrix} \forall x, z \cdot x = nil \wedge z = b \Rightarrow F(x, z) \wedge \\ \forall x, l, z \cdot x \neq nil \wedge F(tail(x), l) \wedge z = l \oplus head(x) \Rightarrow F(x, z) \end{pmatrix} \tag{3}$$

The positive relation constraint corresponding to a catamorphism definition consists of one CHC for the base case and one CHC for the recursive case.

Example 5.4. Consider the axiom for the catamorphism $length$, $\varphi_{(\!|0,+1|\!)}$, from Ex. 4.1. Let $Length$ be a fresh relation symbol. The positive relation constraint for $\varphi_{(\!|0,+1|\!)}$ is:

$$\begin{aligned} &\forall x, z \cdot x = nil \wedge z = 0 \Rightarrow Length(x, z) \wedge \\ \forall x, l, z \cdot x \neq nil \wedge Length(tail(x), l) \wedge z = l + 1 &\Rightarrow Length(x, z) \end{aligned}$$

Despite seemingly capturing only one direction of the definition encapsulated in the catamorphism axiom $\varphi_{(\!|b,\oplus|\!)}$, Thm. 5.7 shows that the positive relation constraint $RC^+_F(\varphi_{(\!|b,\oplus|\!)})$ does not allow spurious solutions when F is used instead of f in C_f, hence preserving satisfiability in both directions. This results from the following lemma that ensures that f is always a solution to $RC^+_F(\varphi_{(\!|b,\oplus|\!)})$ and that all solutions must overapproximate f.

Lemma 5.5 (F over approximates f). *Let M be a model and $\boldsymbol{f}$ an interpretation of f over the universe of M. If $M[f \mapsto \boldsymbol{f}] \models \varphi_{(\!|b,\oplus|\!)}$ then*

a) The relation $\boldsymbol{F}$ over the universe of M defined by the graph of $\boldsymbol{f}$ satisfies $M[F \mapsto \boldsymbol{F}] \models RC^+_F(\varphi_{(\!|b,\oplus|\!)})$.

b) For every relation $\boldsymbol{F}$ over the universe of M, if $M[F \mapsto \boldsymbol{F}] \models RC^+_F(\varphi_{(\!|b,\oplus|\!)})$ then $\boldsymbol{F}$ is an over approximation of the graph of $\boldsymbol{f}$.

[2]The *graph* of a function $\boldsymbol{f}$ is the relation $\boldsymbol{F} = \{(\boldsymbol{x}, \boldsymbol{f}(\boldsymbol{x})) \mid \boldsymbol{x} \in dom(\boldsymbol{f})\}$, where $dom(\boldsymbol{f})$ is the domain of $\boldsymbol{f}$.

Proof. Part *a)* is straightforward. As for part *b)*, let $\text{List}_\mathbf{s}$ denote the set of all elements of sort $\text{List}_\mathbf{s}$ in the universe of M. Wlog, we can assume that $\text{List}_\mathbf{s}$ consists of all constructor terms (see Sec. 3.2). Therefore, we can prove *b)* by straight forward induction on the elements of $\text{List}_\mathbf{s}$: For the base case, since $\boldsymbol{f}(\boldsymbol{nil}) = \boldsymbol{b}$, we have to show that $(\boldsymbol{nil}, \boldsymbol{b}) \in \boldsymbol{F}$. We can see that the clause $\forall x, z \cdot x = nil \wedge b = z \Rightarrow F(x, z)$ enforces this. For the recursive case, assume that $\boldsymbol{f}(\boldsymbol{y}) = \boldsymbol{l}$ and $(\boldsymbol{y}, \boldsymbol{l}) \in \boldsymbol{F}$. Let $\boldsymbol{x} = \boldsymbol{insert}(\boldsymbol{k}, \boldsymbol{y})$ and $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{z}$. It has to be the case that $\boldsymbol{z} = \boldsymbol{k} \oplus \boldsymbol{l}$. The clause $\forall x, l, z \cdot x \neq nil \wedge F(tail(x), l) \wedge z = head(x) \oplus l \Rightarrow F(x, z)$ ensures that $(\boldsymbol{x}, \boldsymbol{z}) \in \boldsymbol{F}$. Thus, all tuples $(\boldsymbol{x}, \boldsymbol{z})$ such that $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{z}$ must belong to the relation $\boldsymbol{F}$. □

We are now ready to define the CHCs obtained after eliminating f:

Definition 5.6 (Relationified CHCs with CHC encoding of catamorphisms). Given C_f and $\varphi_{(\!|b,\oplus|\!)}$, the set of CHCs C_F obtained by relationification of C_f and reducing the function axiom $\varphi_{(\!|b,\oplus|\!)}$ to CHCs is

$$C_F = RC_F^+(\varphi_{(\!|b,\oplus|\!)}) \wedge C_f[f \mapsto F]$$

Theorem 5.7. *The set of CHCs after the transformation is equisatisfiable to the original CHCs modulo catamorphisms:* $C_F \approx \varphi_{(\!|b,\oplus|\!)} \wedge C_f$.

Proof. Left to right direction. Let M be a model for $RC_F^+(\varphi_{(\!|b,\oplus|\!)}) \wedge C_f[f \mapsto F]$, and denote $M[F]$ by $\boldsymbol{F}$. Let $\boldsymbol{f}$ be a function over the universe of M such that $M[f \mapsto \boldsymbol{f}] \models \varphi_{(\!|b,\oplus|\!)}.\boldsymbol{f}$ exists because $\varphi_{(\!|b,\oplus|\!)}$ defines a catamorphism. We know from Lem. 5.5 that $\boldsymbol{F}$ over approximates the graph of $\boldsymbol{f}$. Therefore, by Lem. 5.2 we can see that $M[f \mapsto \boldsymbol{f}]$ is a model for $\varphi_{(\!|b,\oplus|\!)} \wedge C_f$ as well.

Right to Left direction. Let M be a model for $\varphi_{(\!|b,\oplus|\!)} \wedge C_f$. Let $M[f] = \boldsymbol{f}$ and let $\boldsymbol{F}$ be the graph of $\boldsymbol{f}$. By Lem. 5.5, we know that $M[F \mapsto \boldsymbol{F}] \models RC_F^+(\varphi_{(\!|b,\oplus|\!)})$. Therefore, by Lem. 5.2, $M[F \mapsto \boldsymbol{F}] \models C_f[f \mapsto F]$. □

Applying the transformation on CHCs C_f of the form given in Eq. (2), we get:

$$\begin{aligned}
y = nil \wedge z = b &\Rightarrow F(y, z) \\
y \neq nil \wedge F(tail(y), l) \wedge y = l \oplus head(x) &\Rightarrow F(y, z) \\
Init(\bar{y}, \bar{z}) \wedge \big(\bigwedge_i F(y_i, z_i)\big) &\Rightarrow Inv(\bar{y}', \bar{z}') \\
Inv(\bar{y}, \bar{z}) \wedge Tr(\bar{y}, \bar{z}, \bar{y}', \bar{z}') \wedge \big(\bigwedge_i F(y_i, z_i) \wedge F(y_i', z_i')\big) &\Rightarrow Inv(\bar{y}', \bar{z}') \\
Inv(\bar{y}, \bar{z}) \wedge \big(\bigwedge_i F(y_i, z_i)\big) \wedge Bad(\bar{y}, \bar{z}) &\Rightarrow \bot
\end{aligned} \tag{4}$$

While the transformation preserves satisfiability, the following example demonstrates that it may not preserve $\mathcal{A}$-solutions (Def. 3.1):

Example 5.8 (The transformation does not preserve solutions). The set of CHCs in Eq. (5) encode the safety of a program that removes all the elements from a list y while keeping track of its length i. We use term abstraction to encode $length(y)$ via an auxiliary integer variable j, such that neither y nor $length(y)$ are needed in the inductive invariant. Accordingly, the inductive invariant is specified over i, j (note that y is also omitted from Inv in this example). The first clause initializes variable i to the length of y and the second clause keeps track of the length of y as elements are removed from the list. The third clause derives false if i is non zero once all elements are removed.

$$\begin{aligned}
length(y) = i &\Rightarrow Inv(i, i) \\
Inv(i, j) \wedge length(y) = j \wedge y \neq nil \wedge y' = tail(y) \wedge length(y') = j' &\Rightarrow Inv(i - 1, j') \\
Inv(i, j) \wedge length(y) = j \wedge y = nil \wedge i \neq 0 &\Rightarrow \bot
\end{aligned} \tag{5}$$

The set of CHCs in Eq. (5) is satisfiable modulo $\varphi_{(\!|0,+1|\!)}$, the catamorphism axiom for *length*, with the solution: $Inv(i,j) = i = j$, defined over a signature that excludes *length*. The equi-satisfiable CHC $C_F = RC_F^+(\varphi_{(\!|0,+1|\!)}) \wedge C[length \mapsto Length]$ is

$$\begin{aligned} y = nil \wedge z = 0 &\Rightarrow Length(y,z) \\ y \neq nil \wedge Length(tail(y), l) \wedge z = 1 + l &\Rightarrow Length(y,z) \\ Length(y,i) &\Rightarrow Inv(i,i) \\ Inv(i,j) \wedge Length(y,j) \wedge y \neq nil \wedge y' = tail(y) \wedge Length(y',j') &\Rightarrow Inv(i-1,j') \\ Inv(i,j) \wedge Length(y,j) \wedge y = nil \wedge i \neq 0 &\Rightarrow \bot \end{aligned} \tag{6}$$

However, it does not have a solution in which the interpretation for *Length* is expressible without *length*. Intuitively, this is because the interpretation for *Length* in a solution needs to be exactly the graph of the interpretation of *length*, while the relation corresponding to *length* cannot be expressed in first order logic. We formalize this intuition next. Assume to the contrary that there is a model M for the transformed CHCs in which the interpretation of *Length*, $\boldsymbol{Length}$, is not equal to the graph of $\boldsymbol{length}$, where $\boldsymbol{length}$ is the (unique) interpretation of *length* over the universe of M. Then, $\boldsymbol{Length}$ has to be an over approximation of the graph of $\boldsymbol{length}$ (due to Lem. 5.5). Take a pair $(\boldsymbol{y}, \boldsymbol{j})$ that is in $\boldsymbol{Length}$ but not in the graph of $\boldsymbol{length}$, i.e., $\boldsymbol{length}(\boldsymbol{y}) \neq \boldsymbol{j}$, such that $\boldsymbol{y}$ is minimal. We show that the last CHC must be violated. If $\boldsymbol{y} = \boldsymbol{nil}$, use the third clause to deduce that $(\boldsymbol{j}, \boldsymbol{j}) \in \boldsymbol{Inv}$, which immediately implies that the last clause is violated (since by our assumption $\boldsymbol{j} \neq \boldsymbol{0}$ in this case). Otherwise, consider $\boldsymbol{y'} = \boldsymbol{tail}(\boldsymbol{y})$ and $\boldsymbol{j'} = \boldsymbol{length}(\boldsymbol{y'})$. Since $\boldsymbol{Length}$ overapproximates $\boldsymbol{length}$, $(\boldsymbol{y'}, \boldsymbol{j'}) \in \boldsymbol{Length}$ as well. From the 4^{th} clause, we deduce that $(\boldsymbol{j-1}, \boldsymbol{j'}) \in \boldsymbol{Inv}$, even though $\boldsymbol{j-1} \neq \boldsymbol{j'}$ (recall that $\boldsymbol{j} \neq \boldsymbol{length}(\boldsymbol{y})$ whereas $\boldsymbol{length}(\boldsymbol{y}) = \boldsymbol{length}(\boldsymbol{y'}) + \boldsymbol{1} = \boldsymbol{j'+1}$). From $(\boldsymbol{j-1}, \boldsymbol{j'}) \in \boldsymbol{Inv}$, using the pairs in the graph of $\boldsymbol{length}$, we can use the 4^{th} clause to iteratively decrease both elements of the pair by 1, ultimately showing that there exists $\boldsymbol{i} \neq \boldsymbol{0}$ such that $(\boldsymbol{i}, \boldsymbol{0}) \in \boldsymbol{Inv}$. This again contradicts the last clause (by taking the pair $(\boldsymbol{nil}, \boldsymbol{0}) \in \boldsymbol{Length}$).

Remark. We have shown how to relationify catamorphisms on lists. Extending this to catamorphisms on other ADTs is straightforward: relationification of f introduces a Horn clause over the uninterpreted predicate F for each of the base and recursive cases.

The approach also works for RDFs over ADTs that are defined using SMT-LIB syntax. In this case, an RDF is defined by a single axiom: $\forall \bar{v} \cdot f(\bar{v}) = t(\bar{v})$. To perform relationification, we further require that the base and recursive cases can be easily identified. This is already used by the majority of SMT-solvers supporting RDFs. For soundness, we require that the axiom is satisfiable (i.e., actually defines a function). This requirement is omitted from SMT-LIB, but is common in many other systems that allow user-defined recursive functions (e.g., Dafny [Leino 2010]). In general, our approach applies to any RDF that supports an appropriate relationification procedure satisfying Lem. 5.5.

To conclude, in this section, we presented a transformation that takes as input a set of CHCs modulo RDFs, $C_f \wedge \varphi_{(\!|b,\oplus|\!)}$, and produces a set of CHCs without RDFs, C_F. The transformation preserves satisfiability (Thm. 5.7) but not solutions (Ex. 5.8). Note that all solutions to C_F are solutions to $C_f \wedge RC_F^+(\varphi_{(\!|b,\oplus|\!)})$.

6 INTRODUCING RDF ABSTRACTIONS INTO CHCS

In this section we tackle the potential loss of CHC solutions without compromising decidability of the background theory by using abstractions of the RDFs.

We start by re-introducing the RDF with the axiom that defines it, thus recovering solutions. Formally, instead of substituting $f(x) = z$ by $F(x,z)$ during relationification, we substitute it by $f(x) = z \wedge F(x,z)$. Let $C_f[f \mapsto (f \wedge F)]$ denote this transformation.

Definition 6.1 (Relationified CHCs modulo RDFs). Given C_f and $\varphi_{\langle\!|b,\oplus|\!\rangle}$, the set of CHCs C_{fF} obtained after relationification, reducing function axiom $\varphi_{\langle\!|b,\oplus|\!\rangle}$ to CHCs and re-introducing RDFs is

$$C_{fF} = RC_F^+(\varphi_{\langle\!|b,\oplus|\!\rangle}) \wedge C_f[f \mapsto (f \wedge F)]$$

Specifically, when C_f is Eq. (4) modulo $\varphi_{\langle\!|b,\oplus|\!\rangle}$, C_{fF} is,

$$\begin{aligned}
\psi_b(y,z) &\Rightarrow F(y,z)\\
\psi_r(y,z)[f \mapsto F] &\Rightarrow F(y,z)\\
Init(\bar{y},\bar{z}) \wedge \Big(\bigwedge_i F(y_i,z_i) \wedge f(y_i) = z_i\Big) &\Rightarrow Inv(\bar{y}',\bar{z}')\\
Inv(\bar{y},\bar{z}) \wedge Tr(\bar{y},\bar{z},\bar{y}',\bar{z}') \wedge \Big(\bigwedge_i F(y_i,z_i) \wedge f(y_i) = z_i \wedge F(y_i',z_i') \wedge f(y_i') = z_i'\Big) &\Rightarrow Inv(\bar{y}',\bar{z}')\\
Inv(\bar{y},\bar{z}) \wedge \Big(\bigwedge_i F(y_i,z_i) \wedge f(y_i) = z_i\Big) \wedge Bad(\bar{y},\bar{z}) &\Rightarrow \bot
\end{aligned} \tag{7}$$

This transformation preserves not only satisfiability, but also solutions, in the following sense:

Theorem 6.2. *If* $M \models \varphi_{\langle\!|b,\oplus|\!\rangle} \wedge C_f$ *then* $M[F \mapsto \top] \models \varphi_{\langle\!|b,\oplus|\!\rangle} \wedge C_{fF}$. *If* $M \models \varphi_{\langle\!|b,\oplus|\!\rangle} \wedge C_{fF}$, *then* $M \models \varphi_{\langle\!|b,\oplus|\!\rangle} \wedge C_f$.

Proof. First part is straightforward; second part follows from $M[f] \subseteq M[F]$ (Lem. 5.5). □

Corollary 6.3. $\varphi_{\langle\!|b,\oplus|\!\rangle} \wedge C_f \approx \varphi_{\langle\!|b,\oplus|\!\rangle} \wedge C_{fF}$.

From Thm. 6.2 we can see that, if there is an $\mathcal{A}$-solution to $\varphi_{\langle\!|b,\oplus|\!\rangle} \wedge C_f$ then there will also be an $\mathcal{A}$-solution to $\varphi_{\langle\!|b,\oplus|\!\rangle} \wedge C_{fF}$ (assuming that $\top \in \mathcal{A}$).

Unfortunately, re-introducing the catamorphisms with their axioms also makes the underlying logic undecidable. To mitigate this, we work over abstractions of the catamorphism. In the remainder of the section, we define the abstractions we use (Sec. 6.1) and explain what impact the abstractions have on the solutions to the CHCs (Sec. 6.2). In Sec. 7, we describe an algorithm for checking the satisfiability of CHCs modulo RDFs using these abstractions.

6.1 Abstraction of RDF

In this section, we define two abstractions of formulas modulo RDFs. The first abstraction unfolds RDF applications using the axioms of RDF and then replaces any remaining RDF applications with uninterpreted functions and the second abstraction further replaces the literals containing uninterpreted functions with $\top$. Both of the abstractions result in weaker formulas, and can be understood as over approximating the formulas to which they are applied. As we will see later, these abstractions allow us to stay in a decidable fragment when validating solutions, while guaranteeing that we have potentially more solutions than C_F.

To define the abstractions precisely, we first define *prenex* f-pure formulas. A formula is *prenex* f-pure if it is of the form

$$\exists \bar{z} \cdot \Big(\bigwedge_i f(e_i) = z_i\Big) \wedge \theta'(\bar{x},\bar{z}) \tag{8}$$

where each e_i is a term over free variables $\bar{x}$, variables $\bar{z}$, and containing no f, each z_i is a variable in $\bar{z}$, and θ' is a quantifier free formula over $\bar{x}$ and $\bar{z}$ containing no f terms. Every quantifier-free formula θ can be converted into an equivalent prenex f-pure formula by the purification procedure described in Sec. 3: introducing existentially quantified variables that represent the f applications. For example, $p(f(f(x)))$ may be rewritten into $\exists y, z \cdot f(x) = y \wedge f(y) = z \wedge p(z)$.

Instantiating function axiom. Given an RDF f together with its function axiom $\varphi = \forall x \cdot f(x) = t(x)$, and a prenex f-pure formula θ, *instantiation* of f in θ is the formula obtained by conjoining the literal $t[x \mapsto e_i] = z_i$ with each literal $f(e_i) = z_i$ in θ. The *k-instantiation* of f on θ is the formula defined recursively as follows: the 0-instantiation is θ itself and $(i+1)$-instantiation is the instantiation of f on the i-instantiation (after the i-instantiation is brought to prenex f-pure form).

Definition 6.4 (k-instantiation abstraction). Given an RDF f and a prenex f-pure formula θ, the k-instantiation abstraction of f on θ, denoted by $\alpha^k_{rf}(\theta)$, is the formula obtained by replacing all f occurrences in the k-instantiation of θ with a fresh uninterpreted function.

Example 6.5. Let $\theta = length(x) = z_1 \wedge length(y) = z_2 \wedge y = tail(x) \wedge z_1 = 2 \wedge z_2 \neq 1$. Then,

$$\begin{aligned}
\alpha^0_{rf}(\theta) &= length_{uf}(x) = z_1 \wedge length_{uf}(y) = z_2 \wedge y = tail(x) \wedge z_1 = 2 \wedge z_2 \neq 1\\
\alpha^1_{rf}(\theta) &= \alpha^0_{rf}(\theta) \wedge ite(x = nil, 0, 1 + length_{uf}(tail(x))) = z_1 \wedge\\
&\quad\ ite(y = nil, 0, 1 + length_{uf}(tail(y))) = z_2\\
\alpha^2_{rf}(\theta) &= \exists l_{tx}, l_{ty} \cdot length_{uf}(tail(x)) = l_{tx} \wedge length_{uf}(tail(y)) = l_{ty} \wedge\\
&\quad\ \alpha^0_{rf}(\theta) \wedge ite(x = nil, 0, 1 + l_{tx}) = z_1 \wedge ite(y = nil, 0, 1 + l_{ty}) = z_2 \wedge\\
&\quad\ ite(tail(x) = nil, 0, 1 + length_{uf}(tail(tail(x)))) = l_{tx} \wedge\\
&\quad\ ite(tail(y) = nil, 0, 1 + length_{uf}(tail(tail(y)))) = l_{ty}
\end{aligned}$$

Definition 6.6 (Uninterpreted function abstraction). Given an uninterpreted function symbol f, and a prenex f-pure formula θ, the uninterpreted function abstraction, denoted $\alpha^{\uparrow}(\theta)$, is obtained by replacing all literals containing f in θ with $\top$.

We write $\alpha^{k\uparrow}_{rf}(\theta)$ for $(\alpha^{\uparrow}_{rf} \circ \alpha^k_{rf})(\theta)$.

Example 6.7. The result of applying the uninterpreted function abstraction for the k-unrolling abstractions in Ex. 6.5 is

$$\begin{aligned}
\alpha^{0\uparrow}_{rf}(\theta) &= y = tail(x) \wedge z_1 = 2 \wedge z_2 \neq 1\\
\alpha^{1\uparrow}_{rf}(\theta) &= \exists l_{tx}, l_{ty} \cdot \alpha^{0\uparrow}_{rf}(\theta) \wedge ite(x = nil, 0, 1 + l_{tx}) = z_1 \wedge ite(y = nil, 0, 1 + l_{ty}) = z_2\\
\alpha^{2\uparrow}_{rf}(\theta) &= \exists l_{ttx}, l_{tty}, l_{tx}, l_{ty} \cdot \alpha^{0\uparrow}_{rf}(\theta) \wedge ite(x = nil, 0, 1 + l_{tx}) = z_1 \wedge ite(y = nil, 0, 1 + l_{ty}) = z_2 \wedge\\
&\quad\ ite(tail(x) = nil, 0, 1 + l_{ttx}) = l_{tx} \wedge ite(tail(y) = nil, 0, 1 + l_{tty}) = l_{ty}
\end{aligned}$$

Notice that $\alpha^{1\uparrow}_{rf}(\theta)$ has additional, existentially quantified, variables not present in $\alpha^1_{rf}(\theta)$. These variables are introduced during purification. Similarly, the additional variables in $\alpha^{2\uparrow}_{rf}(\theta)$ have also been produced during purification.

The abstractions provide overapproximations of θ in the sense that they may only increase the set of satisfying models:

Lemma 6.8. *Let f be an RDF, θ a prenex f-pure formula and M a model. Then, for any k,*

a) $M \models \theta \implies M[f_{uf} \mapsto M[f]] \models \alpha^k_{rf}(\theta)$, *and*

b) $M \models \alpha^k_{rf}(\theta) \implies M \models \alpha^{k\uparrow}_{rf}(\theta)$.

6.2 CHCs Modulo RDF vs CHCs over RDF Abstractions

Recall that, originally we started with a set of CHCs C_f that contain applications of RDF f. We showed that replacing f with a fresh relation F preserves satisfiability (Thm. 5.7) but not solutions (Ex. 5.8). Re-introducing f regains solutions (Thm. 6.2), but also brings back the problem of

undecidability. In this section, we analyze the effects of re-introducing *abstractions* of the RDF f into C_F. Recall that $C_F = RC_F^+(\varphi_{(\!|b,\oplus|\!)}) \wedge C_f[f \mapsto F]$ is the transformed CHCs without f applications, and $C_{fF} = RC_F^+(\varphi_{(\!|b,\oplus|\!)}) \wedge C_f[f \mapsto (f \wedge F)]$ is the CHCs we get after conjoining back f.

Using the k-instantiation abstraction. When trying to establish that C_{fF} is satisfiable modulo $\varphi_{(\!|b,\oplus|\!)}$, our algorithm uses the k-instantiation abstraction. Let C_{fF}^k be the result of replacing all constraints in the clauses of C_{fF} with their k-instantiation abstractions. (The resulting formulas may be normalized so that all the newly introduced existential quantifiers in the constraints are converted to universal quantifiers over the whole clause.)

C_{fF}^k has uninterpreted functions. This means that satisfiability of C_{fF}^k allows the uninterpreted functions to receive *some* interpretation, while we are interested in solutions that are valid for *any* interpretation of the functions, including the ones that satisfy the abstracted function axioms (akin to an implicit second order universal quantifier over f). Therefore, C_{fF}^k is outside of the usual domain of CHCs, and we refrain from considering its satifiability. Instead, we extend the notion of an $\mathcal{A}$-solution (Def. 3.1) to the abstracted formula C_{fF}^k, and compare $\mathcal{A}$-solutions of C_{fF}^k with $\mathcal{A}$-solutions of the original set of CHCs. Recall that once the solution is plugged in, the requirement is that the resulting sentences are *valid*, which has the effect of implicitly universally quantifying over the uninterpreted function f. The next theorems show that the k-instatiation abstractions induce a hierarchy in terms of their sets of solution.

Theorem 6.9. *Let S be an $\mathcal{A}$-solution to C_{fF}^k. Then, S is an $\mathcal{A}$-solution to C_{fF} modulo $\varphi_{(\!|b,\oplus|\!)}$.*

Proof. Let kC be the set of CHCs obtained by doing k-instantiation (no abstraction) of C_{fF}. By the properties of k-instantiation, we know that $kC \equiv C_{fF}$ modulo $\varphi_{(\!|b,\oplus|\!)}$. Let ψ be the conjunction of the sentences we get from kC by replacing all uninterpreted predicates with their corresponding formulas from S. It suffices to show that ψ is valid modulo $\varphi_{(\!|b,\oplus|\!)}$. Similarly, let ψ^k be the conjunction of sentences we get from C_{fF}^k by replacing all uninterpreted predicates with their corresponding formulas from S. ψ and ψ^k differ only in f being uninterpreted in ψ^k. Since ψ^k is valid, ψ has to be valid modulo $\varphi_{(\!|b,\oplus|\!)}$. Thus, S is an $\mathcal{A}$-solution to C_{fF} modulo $\varphi_{(\!|b,\oplus|\!)}$. □

Importantly, validating $\mathcal{A}$-solutions to C_{fF}^k amounts to checking validity of universally quantified formulas (equivalently, checking unsatisfiability of quantifier-free formulas) modulo the background theory combined with uninterpreted functions, which is decidable.

Another interesting aspect of C_{fF}^k is that $\mathcal{A}$-solutions are preserved as k increases:

Theorem 6.10. *If S is an $\mathcal{A}$-solution to C_{fF}^k, then S is an $\mathcal{A}$-solution to $C_{fF}^{k'}$, where $k' \geq k$.*

Proof. The constraints in the clauses of $C_{fF}^{k'}$ are stronger than the constraints in the corresponding clauses of C_{fF}^k. □

Similarly, if S is an $\mathcal{A}$-solution to C_F, then S is an $\mathcal{A}$-solution to C_{fF}^k for any k. The hierarchy induced by the different abstractions is outlined in Fig. 4. The hierarchy implies that when searching for a solution to C_{fF} modulo $\varphi_{(\!|b,\oplus|\!)}$, we can start with C_{fF}^0 and unroll f more and more, potentially increasing the set of solutions. If we find a solution for any value of k, we have a solution to our original problem.

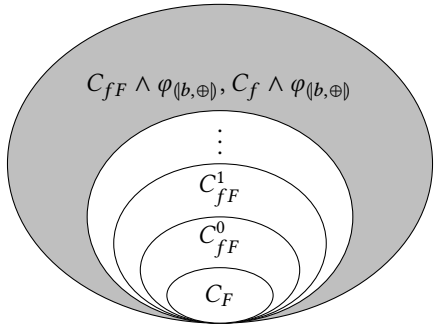

Fig. 4. Comparing $\mathcal{A}$-solutions of different abstractions of CHCs. Each ellipse represents the set of $\mathcal{A}$-solutions to the corresponding formula. Validating solutions of CHC in the shaded region is undecidable.

Example 6.11. Recall the set of CHCs from Ex. 5.8. We showed that the relationified CHC C_F did not have an $\mathcal{A}$-solution. However, the set of abstracted CHCs C^1_{fF}:

$$y = nil \wedge z = 0 \Rightarrow Length(y, z)$$
$$y \neq nil \wedge Length(tail(y), l) \wedge z = 1 + l \Rightarrow Length(y, z)$$
$$Length(y, i) \wedge i = length_{uf}(y) \wedge i = ite(y = nil, 0, 1 + length_{uf}(tail(y))) \Rightarrow Inv(y, i, i)$$
$$Length(y, j) \wedge j = length_{uf}(y) \wedge j = ite(y = nil, 0, 1 + length_{uf}(tail(y))) \wedge$$
$$Length(y', j') \wedge j' = length_{uf}(y') \wedge j' = ite(y' = nil, 0, 1 + length_{uf}(tail(y'))) \wedge$$
$$Inv(y, i, j) \wedge y \neq nil \wedge y' = tail(y) \Rightarrow Inv(y', i - 1, j')$$
$$Length(y, j) \wedge j = length_{uf}(y) \wedge j = ite(y = nil, 0, 1 + length_{uf}(tail(y))) \wedge$$
$$Inv(y, i, j) \wedge y = nil \wedge i \neq 0 \Rightarrow \bot$$

has a solution which maps Inv to the formula $i = j$ (as in the solution for C_f) and $Length$ to the formula $\top$. Furthermore, validating this solution is decidable as the validation query is a quantifier free formula over a combination of ADTs, LIA, and UF.

Using the uninterpreted function abstraction. When trying to establish that C_{fF} is unsatisfiable modulo $\varphi_{(\!|b,\oplus|\!)}$, our algorithm avoids the presence of uninterpreted functions in the CHCs by using the uninterpreted function abstraction.

Let $C^{k\uparrow}_{fF}$ be the result of replacing all constraints in the clauses of C_{fF} with their $\alpha^{k\uparrow}_{rf}$ abstractions. This abstraction preserves counterexamples:

Theorem 6.12. *If $C^{k\uparrow}_{fF}$ is unsatisfiable, then C_{fF} is unsatisfiable modulo $\varphi_{(\!|b,\oplus|\!)}$.*

Proof. We know from Thm. 5.7 and Thm. 6.2 that, modulo $\varphi_{(\!|b,\oplus|\!)}$, C_F is equisatisfiable to C_{fF}. We prove that if $C^{k\uparrow}_{fF}$ is unsatisfiable, then C_F is unsatisfiable. The contrapositive of this statement is that if C_F is satisfiable, then $C^{k\uparrow}_{fF}$ is satisfiable. The constraints in the clauses of $C^{k\uparrow}_{fF}$ are stronger than the constraints in the corresponding clauses of C_F. Therefore, for all models M, if $M \models C_F$, then $M \models C^{k\uparrow}_{fF}$. □

7 SOLVING CHC MODULO ADT AND RDF

In this section, we present Racer, an algorithm that solves CHCs of the form C_{fF} (Eq. (7)) modulo a catamorphism axiom $\varphi_{(\!|b,\oplus|\!)}$ for f. Racer is sound: if it finds a solution, it is a solution for the original set of CHCs, C_f, modulo the axiom; if it declares UNSAT, then the original set of CHCs

modulo the axiom is UNSAT. Further, if C_f is UNSAT, Racer is guaranteed to terminate. Racer works by simultaneously searching for $\mathcal{A}$-solutions of C^k_{fF} as well as counter examples to $C^{k\uparrow}_{fF}$ while periodically increasing k. Racer is based on Spacer, an IC3-style algorithm for solving CHCs. We first give background on Spacer and explain necessary notation (Sec. 7.1) and then explain Racer (Sec. 7.2).

7.1 Background on Spacer

Spacer is based on IC3, which was developed for safety Model Checking. For that reason, we switch terminology from logic to *states* and *transitions* (as explained in Sec. 3.3), instead of adapting IC3 to the terminology of the paper. Our implementation, however, works with general CHCs.

Spacer works by generating and blocking predecessors to *Bad* states called *proof obligations* (POB). Each time Spacer blocks a POB, it learns a *lemma* that blocks many similar POBs. Each time Spacer proves that a POB is reachable from the initial states, it computes *reachable states* to avoid generating the same POB again. For each predicate, the lemmas can be used to construct a *may summary* whereas the reachable states can be used to create a *must summary*. The may summaries are used to create $\mathcal{A}$-solutions.

Alg. 1 describes our contributions (highlighted) on top of the Spacer algorithm. To make the presentation concise, we have used the following notation.

Notation. All formulas in the algorithm are over state variables $\bar{x}$. For a formula θ over state variables, we use the shorthand $\theta[\bar{y}]$ to mean $\theta[\bar{x} \mapsto \bar{y}]$. We assume that whenever a predicate $\mathcal{P}$ appears in the head of a Horn clause, it has the same variables as its arguments. We use the notation $\overline{\text{UT}}^{\mathcal{P}}$ to represent the *uninterpreted tail* of predicate $\mathcal{P}$: the sequence of all pairs $\langle Q, \bar{v}\rangle$ where Q is an uninterpreted predicate and the literal $Q(\bar{v})$ appears in the tail of a Horn clause whose head is $\mathcal{P}$. We use the notation $\overline{\text{UT}}^{\mathcal{P}}_i$ to mean the uninterpreted tail of the ith Horn clause whose head is $\mathcal{P}$. As a special case, we use $\overline{\text{UT}}^{Bad}$ to refer to the set of pairs of uninterpreted predicates and the argument variables in the tail of the Horn clause whose head is *Bad*. Let $\mathcal{Y}$ be a map from uninterpreted predicates to their summaries, where a summary is a formula over $\bar{x}$, and $\mathcal{Y}^Q$ is the summary of uninterpreted predicate Q in $\mathcal{Y}$. We use the shorthand $\mathcal{F}_{\mathcal{P}}(\mathcal{Y})$ to denote the *predicate transformer* of $\mathcal{P}$ defined as:

$$\mathcal{F}_{\mathcal{P}}(\mathcal{Y}) = (Init^{\mathcal{P}})' \vee \bigvee_i \left(\left(\bigwedge_{\langle Q,\bar{v}\rangle \in \overline{\text{UT}}^{\mathcal{P}}_i} \mathcal{Y}^Q[\bar{x} \mapsto \bar{v}] \right) \wedge Tr_i \right)$$

where $Init^{\mathcal{P}}$ is the disjunction of tails of Horn clauses which have no uninterpreted predicates in their tails and whose head is $\mathcal{P}$ and Tr_i is the constraint in the ith Horn clause whose head is $\mathcal{P}$. We use R to denote the map from uninterpreted predicates to their must summaries. Similarly, we use O_i to denote the map from uninterpreted predicates to their may summaries in the i'th frame. We denote the sequence $O_0, O_1, \ldots$ by $\bar{O}$, and write $\bar{O}_i$ to refer to O_i.A POB is represented as a triple $\langle \theta, i, \mathcal{P}\rangle$, where θ is formula in List_s + r (List_s is the sort of List and r is the return sort of RDF), i is a natural number representing the level at which θ is to be blocked and $\mathcal{P}$ denotes the uninterpreted predicate to which the POB belongs. For CHCs like those in Eq. (7), $\mathcal{P}$ can either be I, the uninterpreted predicate representing the desired inductive invariant of the program or F, the relationified RDF.

7.2 The Racer Algorithm

We are ready to explain the full algorithm. The input to Racer (Alg. 1) are CHCs of the form Eq. (7) together with a function axiom for f and the output is SAFE/UNSAFE. Racer need not terminate.

Algorithm 1: The RACER algorithm. The parts of the algorithm which are highlighted in green (resp. yellow) are the only places that use α^k_{rf} (resp. $\alpha^{k\uparrow}_{rf}$) abstraction. Pink highlights the increment of k. The rest of the algorithm is the same as SPACER.

Input: A CHC of the form Eq. (7)
Input: A catamorphism axiom $\varphi_{\langle b,\oplus\rangle}$
Output: SAT/UNSAT

1 $R^F := \{(nil, b)\}$;
2 $\bar{O}^F_0 = R^F; \bar{O}^F_i := \top, \forall i > 0$
3 $R^I := \bot; \bar{O}^I_i := \top, \forall i \geq 0$;
4 $N := 0; k := 0$
5 create_pred_bad()
6 **while** $\top$ **do**
7 $\langle \theta, i+1, \mathcal{P}\rangle := Q.pop()$
8 $\mathcal{I}, M, res :=$ check_reach($\langle \theta, i, \mathcal{P}\rangle$)
9 **if** res **then**
10 create_pred($\mathcal{I}, \langle \theta, i+1, \mathcal{P}\rangle, M$)
11 **if** is_bad_rch() **then return** UNSAFE
12 **else**
13 block-pob($\langle \theta, i+1, \mathcal{P}\rangle$)
14 $Q := Q \cup \langle \theta, i+2, \mathcal{P}\rangle$
15 **if** is_bad_blkd() **then**
16 $k := k + 1$
17 $N := N + 1$
18 create_pred_bad()
19 **if** chk_ind() **then return** SAFE
20 **function** *chk_ind()*:
21 **return** $\exists j \cdot 0 \leq j \leq N-1 \wedge$
22 $\bar{O}^I_{j+1} \Rightarrow \bar{O}^I_j \wedge \bar{O}^F_{j+1} \Rightarrow \bar{O}^F_j$
23 **function** *check-reach*($\langle \theta, i, \mathcal{P}\rangle$):
24 $\beta := \alpha^k_{rf}(\mathcal{F}_{\mathcal{P}}(\bar{O}_i)) \wedge \varphi'$
25 **if** $\neg$*isSAT*(β) **then return** $\_, \_, \bot$
26 $\mu := \{R^Q[\bar{v}] \mid \langle Q, \bar{v}\rangle \in \overline{\text{UT}}^{\mathcal{P}}\}$
27 $M :=$ max_sat(β, μ)
28 $\mathcal{I} :=$ Implicant(β, M)
29 $\mathcal{I} := \mathcal{I} \wedge \{r \mid r \in \mu \wedge M \models r\}$
30 **return** $\mathcal{I}, M, \top$
31 **function** *is_bad_rch()*:
32 $\theta := R^I \wedge (\bigwedge_i R^F[y_i, z_i]) \wedge Bad$
33 **return** isSAT(θ)
34 **function** *block_pob*($\langle \theta, i+1, \mathcal{P}\rangle$):
35 $\beta := \alpha^k_{rf}(\mathcal{F}_{\mathcal{P}}(\bar{O}_i))$
36 $c :=$ ITP(β, θ')[$\bar{x}' \mapsto \bar{x}$]
37 $\ell :=$ IND_GEN(c)
38 $\bar{O}^{\mathcal{P}}_j := \bar{O}^{\mathcal{P}}_j \wedge \ell \quad \forall j \leq i+1$
39 **function** *create_pred_bad()*:
40 $\theta := \bar{O}^I_{N+1} \wedge (\bigwedge_i \bar{O}^F_{N+1}[y_i, z_i] \wedge \alpha^{k\uparrow}_{rf}(f(y_i) = z_i)) \wedge Bad$
41 **if** $\neg$*isSAT*(θ) **then return**
42 Let $M \models \theta; \bar{v} := vars(\theta)$
43 **for** $\langle Q, \bar{y}\rangle \in \overline{UT}^{Bad}$ **do**
44 $pob :=$ MBP($\bar{v} \setminus \{\bar{y}\}, \theta, M$) $Q := Q \cup \langle pob, N+1, Q\rangle$
45 **function** *create_pred*($\mathcal{I}, \langle \theta, i+1, \mathcal{P}\rangle, M$):
46 $\Phi := \{\langle Q, \bar{u}\rangle \mid \langle Q, \bar{u}\rangle \in \overline{\text{UT}}^{\mathcal{P}} \wedge M \not\models R^Q[\bar{u}]\}$
47 **if** $\Phi = \emptyset$ **then**
// compute successor
48 $\bar{v} :=$ vars($\mathcal{I}$) $\setminus \{\bar{x}'\}$
49 $r :=$ MBP($\bar{v}, \alpha^{\uparrow}_{rf}(\mathcal{I} \wedge \theta'), M$)
50 $R^{\mathcal{P}} := R^{\mathcal{P}} \vee r[\bar{x}' \mapsto \bar{x}]$
51 **else**
52 $kids := \emptyset$
53 **for** $\langle Q, \bar{u}\rangle \in \Phi$ **do**
54 $\bar{v} :=$ vars($\mathcal{I}$) $\setminus \{\bar{u}\}$
55 $\gamma :=$ MBP($\bar{v}, \alpha^{\uparrow}_{rf}(\mathcal{I} \wedge \theta'), M$)
56 $\gamma := \gamma[\bar{u} \mapsto \bar{x}]$
57 $kids := kids \cup \langle \gamma, i, Q\rangle$
58 $Q := Q \cup kids$
59 $Q := Q \cup \langle \theta, (i+1), \mathcal{P}\rangle$
60 **function** *is_bad_blkd()*:
61 $\theta := \bar{O}^I_N \wedge (\bigwedge_i \bar{O}^F_N[y_i, z_i] \ \wedge \ \alpha^k_{rf}(f(y_i) = z_i)) \wedge Bad$
62 **return** $\neg$ isSAT(θ)

The Initialization phase. RACER initializes both R^F and $\bar{O}^F_0$ to $\{(nil, b)\}$ (Line 1). Note that we are using a set of states to represent a formula over variables $\bar{x}$. Next, R^I is initialized with $\bot$ and all the $\bar{O}^I$ with $\top$. These initialization steps are sound because in Eq. (7), F is the only predicate that appears in the head of a clause that has no predicate in the body. After initialization, RACER creates predecessors to *Bad* and add them to the queue (line 5).

IC3-part of the algorithm. In each iteration RACER calls the *check_reach* function to check the reachability of a POB at a particular level (line 8). The *check_reach* function first checks whether the POB is blocked using may summaries (line 25). If the check is satisfiable, the function finds a model that satisfies as many must summaries as possible using a *max_sat* algorithm. The *check_reach*

function returns $\top$ only if the may summaries are not strong enough to block the POB at this level. In this case, the algorithm calls the *create_pred* function to create and add predecessor POBs to the queue (line 10). Otherwise, it calls the *block_pob* to learn a lemma that blocks the POB (line 13). The *create_pred* function (line 45) uses *Model Based Projection (MBP)* [Komuravelli et al. 2016] to create a predecessor POB for all the predicates in the tail whose must summaries are *not* satisfied by the model (but their may summaries are). If there are no such predicates, the POB is reachable from the must summaries. In this case, the algorithm computes a reachable state that intersects with the POB using MBP (line 49) and adds it to the set of reachable states. Racer routinely checks whether *Bad* is reachable (line 11) and increases the depth of exploration when *Bad* is blocked (line 15).

The role of abstractions. When checking for one-step reachability, Racer uses the k-instantiation abstraction to abstract the predicate transformer (line 24). Racer employs the same abstraction when learning a lemma (line 34) and when checking whether *Bad* is blocked (line 60). This ensures any inductive invariant Racer finds is an $\mathcal{A}$-solution to C^k_{fF}. Racer employs both uninterpreted function abstraction and k-instantiation abstraction when computing predecessors (line 55) and successors (line 49). This ensures that if *Bad* is reachable, it is reachable in $C^{k\uparrow}_{fF}$.

MBP. Racer generates predecessor POBs by using *Model Based Projection* (MBP) [Komuravelli et al. 2016] to eliminate post state variables. Given a quantifier free formula θ, a model $M \models \theta$, and a set of constants $\bar{x}$, $MBP(\theta, \bar{x}, M)$ is a model preserving under-approximation of θ projected onto $\Sigma \setminus \bar{x}$. That is, $MBP(\theta, \bar{x}, M)$ is a quantifier free formula ψ such that (1) ψ does not contain the constants $\bar{x}$, (2) constants of ψ are a subset of constants of θ, and (3) ψ under-approximates the result of eliminating $\bar{x}$ from θ: $\psi \Rightarrow (\exists \bar{x} \cdot \theta)$. While there are algorithms to compute MBP for the theory of ADTs [Bjørner and Janota 2015], it is not clear how to compute MBP in the presence of UF because of lack of quantifier elimination. The abstractions ensure that there are no UF when MBP is applied. Thus, Racer uses MBP for ADT+LIA from [Bjørner and Janota 2015].

Progress. From the properties of IC3, it follows that, whenever Racer or Spacer increments the depth of exploration (N in Alg. 1), there are no counterexamples of lower depth. We say that Racer (Spacer) makes *progress* if all non-terminating executions of Racer (Spacer) increment the depth of exploration infinitely often. In Racer (Alg. 1), this amounts to executing line 15 infinitely often on all non-terminating executions.

Spacer makes progress as long as the underlying theory is decidable and has a finite MBP. Racer keeps the underlying theory decidable by using abstractions to ensure that all queries are quantifier free formulas without RDFs. However, unlike Spacer, Racer does not use the same formula to check reachability of a POB and to compute its predecessors. Instead, Racer uses only the k-instantiation abstraction when checking reachability and uses both k-instantiation abstraction as well as uninterpreted function abstraction when computing predecessors using MBP. This makes arguing about progress a little tricky.

We first make two assumptions and show that Racer makes progress under those assumptions. We then relax those assumptions and argue why Racer still makes progress. Assume that k is fixed and that, during the one-step reachability check Racer employs both uninterpreted function abstraction as well as the k-instantiation abstraction instead of doing only the k-instantiation abstraction. Therefore, just like in the original Spacer, the formula to check reachability of a POB and the formula to compute its predecessors are the same. Since the MBP for the underlying theory is finite [Bjørner and Janota 2015], each POB can only have a finite number of predecessors. Therefore, Racer makes progress under these assumptions. Now we relax the second assumption and employ *only* the k-instantiation abstraction during one-step reachability check. Let $\alpha^k_{rf}(\theta)$ be

the k-instantiation abstracted formula that Racer uses to check reachability. We know that $\alpha_{rf}^{k\uparrow}(\theta)$ is weaker than $\alpha_{rf}^{k}(\theta)$. Therefore, all models of $\alpha_{rf}^{k}(\theta)$ are also models of $\alpha_{rf}^{k\uparrow}(\theta)$. Hence, when Racer computes MBP, no new models have been introduced just because we relaxed this assumption. Therefore, MBP is still be finite and Racer still makes progress. As for the first assumption, Racer makes progress as long as k is not increased too frequently. In our implementation, k is increased by 1 each time N is increased. Since the depth of counterexamples in any unsatisfiable CHCs is finite, the progress guarantee ensures that Racer is guaranteed to terminate on all unsatisfiable CHCs. Combining this with the results from earlier sections, we conclude the following:

Theorem 7.1. *Let C_{fF} be a set of reltionified CHCs modulo function axiom $\varphi_{(\!|b,\oplus|\!)}$. If C_{fF} is UNSAT, Racer terminates and returns UNSAT. If C_{fF} is satisfiable, if Racer terminates, it returns a solution to C_{fF}.*

8 IMPLEMENTATION AND EVALUATION

We have implemented Racer on top of Spacer in Z3[3]. Our implementation supports CHCs modulo multiple ADTs as well as multiple RDFs as long as all RDFs are relationified (i.e., in the form Eq. (7)).

Benchmark description. While we are mainly interested in solving CHCs encoding imperative programs, to show the robustness of our approach, we evaluate Racer on CHC benchmarks obtained from two sources. The first group, called Rust-bench, contains 87 CHCs encoding safety of imperative Rust programs from [Matsushita et al. 2020]. The second group, called Leon-bench, contains 64 CHCs encoding safety of ADT manipulating functional programs [De Angelis et al. 2020].

Getting benchmarks into the desired form. While the benchmarks encode programs manipulating ADTs, they do not take advantage of RDFs. That is, the CHCs in these benchmarks capture programs using only uninterpreted predicates. They are relationified CHCs conjoined with positive relation constraints of RDFs (of the form Eq. (4), extended to include multiple F predicates).

We convert these CHCs into the desired form[4] of C_{fF} (Eq. (7)) modulo all function axioms by identifying some predicates F with the RDFs f they correspond to, and recovering their RDF definitions. Specifically, we identify a subset of program methods as implementing RDFs and all other program methods (including recursive ones) as an implementation of some imperative program. For all program methods that define RDFs, we produce both their CHC encoding (positive relation constraint) and RDF axioms. All program methods that compute arithmetic properties of ADTs are considered as RDFs. For example, a method that checks whether a given list contains an element is considered an RDF, whereas a method that appends a given element to a list is not.

We use *term abstraction* to facilitate solutions that do not contain RDF applications. Specifically, we introduce new (arithmetic) variables to represent applications of RDFs that are necessary to prove safety. These variables are added to predicates in the CHCs as arguments that can be used in solutions. The equalities between the variables and the RDF applications that they represent are added to the bodies of the CHCs. Term abstraction is done greedily: for each RDF and each predicate argument, if the RDF can be applied either to the argument itself or, if the argument is a record type, to a selector applied to the argument, a new variable is introduced to capture the RDF application. We stress that this process is completely automatic. That is, we use the same exact input for all the tools in our evaluation. For our tool, we do a simple preprocessing step that identifies relationfied RDFs, re-encodes them as RDFs, and generates terms for term abstraction.

[3]Our implementation is available at https://github.com/hgvk94/z3/tree/racer

[4]Both modified and unmodified benchmarks are publicly available at https://doi.org/10.5281/zenodo.5083780.

Table 1. Comparing solved instances on Rust-bench. In each cell, the first entry is the number of SAT instances solved and the second entry is the number of UNSAT instances solved in that category.

Category	Count	Eldarica	Spacer	HoIce	Racer
Programs with RDFs	44	0/0	1/8	18/3	36/0
Programs without RDFs	43	7/7	22/20	13/15	22/20
Total	87	7/7	23/28	31/18	58/20

Evaluation on Rust-bench. The Rust-bench benchmark suite encodes Rust programs that manipulate ADTs. Of the 87 benchmarks, only 44 manipulate recursive ADTs (lists and trees) and contain (recovered) RDFs. The RDFs computed the length of lists, size of trees and sum of elements of trees and lists. On these benchmarks, we compare Racer with Eldarica[5] [Hojjat and Rümmer 2018], Spacer, and HoIce— state-of-the-art tools that support CHCs modulo ADT. Since none of these solvers support CHCs modulo RDFs, they cannot be run on our modified benchmarks. To make a comparison, we run them on the original set of benchmarks from [Matsushita et al. 2020]. We ran all four solvers with a timeout of 100s. Our results are shown in Tab. 1. In total, Racer solves 29 (resp. 27) more benchmarks than HoIce (resp. Spacer). When we only consider programs that contain RDFs, Racer solves 15 (resp. 27) more instances than HoIce (resp. Spacer). The superiority of Racer is clear when looking at the number of SAT instances it solves (58) compared to HoIce (31) and Spacer (23).

Evaluation on Leon-bench. The benchmarks in Leon-bench encode verification condition validation queries generated by Leon [Suter et al. 2011]. The benchmarks encode operations on three types of ADTs: Queues, Trees, and Heaps. The RDFs compute size of the ADTs, checks membership in them, removes the last element, and checks whether they are sorted. We present our evaluation results on SAT and UNSAT CHCs separately. We further split the SAT benchmarks based on the type of ADTs.

On Leon-bench, we compare Racer with three CHC solvers: VeriMap [De Angelis et al. 2020], Spacer, Eladarica and with CVC4 [Reynolds and Kuncak 2015] (with the subgoal generation heuristic enabled). For our evaluation, we use three different versions of these benchmarks: (1) to evaluate Verimap, Eldarica, and Spacer, we use the CHC encoding over ADTs and LIA, (2) to evaluate CVC4, we use the original verification condition validation queries but without any of the subgoals, as done in [De Angelis et al. 2020], and (3) to evaluate Racer, we conjoin RDFs to CHCs as mentioned earlier.

Tab. 2 shows the results of our evaluation. In summary, Racer is quite competitive with all other solvers even though Racer was designed for verifying imperative programs and not functional programs. In fact, in almost all categories, Racer solves benchmarks that other solvers do not solve.

We could not compare Racer with all solvers on all benchmarks because the tools do not support exactly the same inputs: Eldarica, HoIce, and Spacer support CHCs in the SMT2-LIB format, VeriMap supports CHCs modulo ADTs but in the Prolog format, and the inductive reasoning in CVC4 works with SMT-LIB formulas. We also repeated the experiments with a larger timeout of 30 minutes and did not see any change in number of instances solved for any of the solvers.

[5]We could not make use of Eldarica's support for size constraints because in our benchmarks, RDFs are not restricted to size constraints.

Table 2. Comparing Racer with Spacer, Eldarica, CVC4, and VeriMAP on Leon-bench. The first three rows show SAT instances. The **UNSAT** row shows UNSAT instances from all three categories. The number of unique instances solved are shown in brackets.

Category	Count	Eldarica	VeriMAP	CVC4	Spacer	Racer
Queue	11	2	4 (2)	3	0	4 (1)
Tree	18	0	12 (1)	7 (1)	0	7
Heap	12	0	7 (3)	1	0	4 (1)
UNSAT	23	20	13	0	23 (1)	22
Total	64	22	36	11	23	37

9 RELATED WORK

The theory of ADTs and RDFs is part of the SMT-LIB v2.6 standard [Barrett et al. 2017] and is well supported by the major SMT solvers. Most decision procedures for ADTs are based on finite instantiation of ADT axioms [Barrett et al. 2007; Bjørner 1999; Oppen 1980]. The combination of ADTs and RDFs is undecidable. Most existing solvers use finite unrollings of RDF definitions to check satisfiability [Suter et al. 2010]. This is similar to how quantifiers are typically handled. The solver returns UNSAT if some finite unrolling of RDFs is sufficient for unsatisfiability, otherwise, the solver might return unknown. This is usually not a significant hurdle for deductive-style verification tools that treat *unknown* as unproven and rely on the user to supply additional lemmas to help the solver. At the same time, this is a serious issue for all automated verification tools since they rely on the solver's SAT answers to drive abstraction refinement. We overcome this problem by unrolling RDFs and then either abstracting the result with uninterpreted functions or abstracting RDFs altogether. In both cases, the abstract formulas are decidable and we use their models to drive the abstraction-refinement process within Racer.

In some cases, extending ADTs with RDFs remains decidable. A particularly interesting class of decidable RDFs is called (generalized) infinitely surjective catamorphisms [Pham et al. 2016; Suter et al. 2010, 2011]. The class includes many useful RDFs over ADTs including length of a list, height and size of a tree, as well as various filters over lists and trees. However, showing that a given RDF is an infinitely surjective catamorphism is not easy, which limits their application in a completely automated procedure such as Racer. Interestingly, in our running example (Fig. 1), Racer is able to automatically generate the necessary property of the `length` function to show that it is infinitely surjective, and, with this lemma, SMT queries involving `length` become decidable. That is, the existing procedure of finite unfolding terminates with either SAT or UNSAT.

More specialized restrictions, such as restricting to specific catamorphisms have been considered as well [Hojjat and Rümmer 2017; Zhang et al. 2006]. In contrast, we do not restrict the definition of the catamorphism, and automatically infer supporting lemmas for surjectivity. Moreover, our approach does not differentiate between catamorphisms and arbitrary RDFs. Of course, our procedure in incomplete and need not terminate.

Prior to inclusion of RDFs in the SMT-LIB standard and their native support in solvers, deductive verification tools have shown that dynamic unfolding of RDFs can be simulated by quantifier instantiation. Most prominent is the so called *fuel* approach of Amin et al. [Amin et al. 2014] that uses ADTs to encode ordinals to control number of unrollings of RDFs. Current state-of-the-art procedures for RDFs work directly within an SMT solver and unroll RDFs by adding (and removing) constraints to the SMT solver using its incremental interface. This is also what we do in our

implementation. One of the advantages is that this enables combining RDFs with other theories and without enabling quantifier support in the solver. We do not compare with [Amin et al. 2014], or any deductive-style verification approach, since they require the user to provide candidate inductive invariants while Racer is completely automatic.

Automatic inductive reasoning is studied in the context of SMT (e.g., [Reynolds and Kuncak 2015]), automated theorem proving (e.g., [Reger and Voronkov 2019]), and deductive program analysis (e.g., [Leino 2012]). However, these works do not apply directly to CHC-solving, and especially in combination with other theories such as LIA. For example, the TIP benchmarks [Rosén and Smallbone 2015] focus on functional programs and model integers with natural numbers.

There are a number of CHC Solvers that support ADT constraints. VeriMAP [De Angelis et al. 2018] transforms CHCs with ADTs to CHCs over basic types, by using folding/unfolding and by introducing difference predicates [De Angelis et al. 2020]. In this way, VeriMAP separates ADT reasoning from CHC solving. VeriMAP can determine satisfiability of CHCs but cannot generate solutions. Most importantly, VeriMAP is unsound for UNSAT answers. In contrast, our approach is sound for both SAT and UNSAT and generates solutions that can be independently verified. CHC solver HoIce [Champion et al. 2018] can produce solutions containing RDFs (but it does not allow RDFs in the input). It is based on the ICE framework that iteratively enumerates the space of solutions including a restricted set of RDFs. In contrast to Racer, HoIce does not provide progress guarantees nor always generates a counterexample. The RInGen [Kostyukov et al. 2021] solver infers solutions to CHCs modulo ADTs by treating all symbols as uninterpreted (thus, it does not support arithmetic). It uses a finite model finder to infer a finite model for the resulting FoL formula and extends it to a solution expressed using a tree automaton. Since RInGen treats all symbols as uninterpreted, it does not support types and could not be compared to our approach.

In SMT, it is common to abstract complex functions (e.g., multiplication, RDFs) by uninterpreted functions. However, lifting this abstraction effectively to CHC is still an open challenge. One issue is that in the context of CHCs, uninterpreted functions are *universally* quantified. That is, the solver must find a solution that works for *all* interpretaions of UFs, not synthesize a model for these functions. An interested reader is referred to [Bjørner et al. 2015] for the formal definition. We avoid dealing with UFs by requiring that all functions are explicitly defined and UFs are only used as local abstraction during the algorithm. This is similar to support for UFs in Euforia [Bueno and Sakallah 2019], except that we deal with recursively defined functions rather than bit-vector terms.

In this paper, we present a procedure for solving CHCs modulo ADTs and RDFs. Our approach sidesteps the undecidability of the underlying logic by simultaneously approximating the RDFs using multiple abstractions. One abstraction compiles RDF to CHCs while preserving satisfiability (but not solutions). It is used to detect unsatisfiability of CHCs (i.e., counterexamples to safety verification). The other abstraction, parameterized by the depth of unfolding k, replaces RDFs by their finite unfolding. As k increases, the abstraction enables more solutions that are potentially lost by the first abstraction. The two abstractions are explored in tandem in an IC3-style algorithm. Remarkably, the algorithm is able to automatically combine learning inductive invariants over ADTs and RDFs with learning inductive lemmas of RDFs.

While our presentation focuses on specific RDFs – catamorphisms – our results are more general. They apply to arbitrary well defined RDFs. The only requirements are that the function is total and it is possible to isolate the base- and recursive-cases in the definition.

We have implemented our technique in the Spacer CHC-solver of Z3. Our evaluation shows significant improvement of our technique over state-of-the-art CHC solvers on imperative programs while still being competitive on functional programs.

In this work, we assumed that RDFs are given (or mined) in the original problem. It is interesting to extend the work further by synthesizing RDFs together with an inductive invariant. We believe

that many simple catamorphisms, such as length, size, height, etc., and their variants, can be constructed automatically during the search. We leave exploring this challenging direction to future work.

ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement No [759102-SVIS]). This research was partially supported by the United States-Israel Binational Science Foundation (BSF) grant No. 2016260, and the Israeli Science Foundation (ISF) grant No. 1810/18. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) and MathWorks Inc.

REFERENCES

Francesco Alberti, Roberto Bruttomesso, Silvio Ghilardi, Silvio Ranise, and Natasha Sharygina. 2012. Lazy Abstraction with Interpolants for Arrays. In *Logic for Programming, Artificial Intelligence, and Reasoning - 18th International Conference, LPAR-18, Mérida, Venezuela, March 11-15, 2012. Proceedings (Lecture Notes in Computer Science)*, Nikolaj Bjørner and Andrei Voronkov (Eds.), Vol. 7180. Springer, 46–61. https://doi.org/10.1007/978-3-642-28717-6_7

Nada Amin, K. Rustan M. Leino, and Tiark Rompf. 2014. Computing with an SMT Solver. In *Tests and Proofs - 8th International Conference, TAP@STAF 2014, York, UK, July 24-25, 2014. Proceedings (Lecture Notes in Computer Science)*, Martina Seidl and Nikolai Tillmann (Eds.), Vol. 8570. Springer, 20–35. https://doi.org/10.1007/978-3-319-09099-3_2

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`.

Clark W. Barrett, Igor Shikanian, and Cesare Tinelli. 2007. An Abstract Decision Procedure for a Theory of Inductive Data Types. *J. Satisf. Boolean Model. Comput.* 3, 1-2 (2007), 21–46. https://doi.org/10.3233/sat190028

Nikolaj Bjørner. 1999. *Integrating Decision Procedures for Temporal Verification.* Ph.D. Dissertation. Stanford University, Stanford, CA, USA. Advisor(s) Manna, Zohar. AAI9924398.

Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. 2015. Horn Clause Solvers for Program Verification. In *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday (Lecture Notes in Computer Science)*, Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner, and Wolfram Schulte (Eds.), Vol. 9300. Springer, 24–51. https://doi.org/10.1007/978-3-319-23534-9_2

Nikolaj Bjørner and Mikolás Janota. 2015. Playing with Quantified Satisfaction. In *20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning - Short Presentations, LPAR 2015, Suva, Fiji, November 24-28, 2015 (EPiC Series in Computing)*, Ansgar Fehnker, Annabelle McIver, Geoff Sutcliffe, and Andrei Voronkov (Eds.), Vol. 35. EasyChair, 15–27.

Denis Bueno and Karem A. Sakallah. 2019. EUForia: Complete Software Model Checking with Uninterpreted Functions. In *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings (Lecture Notes in Computer Science)*, Constantin Enea and Ruzica Piskac (Eds.), Vol. 11388. Springer, 363–385. https://doi.org/10.1007/978-3-030-11245-5_17

Adrien Champion, Tomoya Chiba, Naoki Kobayashi, and Ryosuke Sato. 2018. ICE-Based Refinement Type Discovery for Higher-Order Functional Programs. In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I (Lecture Notes in Computer Science)*, Dirk Beyer and Marieke Huisman (Eds.), Vol. 10805. Springer, 365–384. https://doi.org/10.1007/978-3-319-89960-2_20

CHC-COMP. 2021. CHC-COMP. https://chc-comp.github.io.

Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. 2018. Solving Horn Clauses on Inductive Data Types Without Induction. *Theory Pract. Log. Program.* 18, 3-4 (2018), 452–469. https://doi.org/10.1017/S1471068418000157

Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. 2020. Removing Algebraic Data Types from Constrained Horn Clauses Using Difference Predicates. In *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I (Lecture Notes in Computer Science)*, Nicolas Peltier and Viorica Sofronie-Stokkermans (Eds.), Vol. 12166. Springer, 83–102. https://doi.org/10.1007/978-3-030-51074-9_6

Grigory Fedyukovich, Samuel J. Kaufman, and Rastislav Bodík. 2017. Sampling invariants from frequency distributions. In *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, Daryl Stewart and Georg Weissenbacher (Eds.). IEEE, 100–107. https://doi.org/10.23919/FMCAD.2017.8102247

Grigory Fedyukovich, Samuel J. Kaufman, and Rastislav Bodík. 2020. Learning inductive invariants by sampling from frequency distributions. *Formal Methods Syst. Des.* 56, 1 (2020), 154–177. https://doi.org/10.1007/s10703-020-00349-x

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 - Where Programs Meet Provers. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science)*, Matthias Felleisen and Philippa Gardner (Eds.), Vol. 7792. Springer, 125–128. https://doi.org/10.1007/978-3-642-37036-6_8

Cormac Flanagan, Rajeev Joshi, and K. Rustan M. Leino. 2001. Annotation inference for modular checkers. *Inf. Process. Lett.* 77, 2-4 (2001), 97–108. https://doi.org/10.1016/S0020-0190(00)00196-4

Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an Annotation Assistant for ESC/Java. In *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001, Proceedings (Lecture Notes in Computer Science)*, José Nuno Oliveira and Pamela Zave (Eds.), Vol. 2021. Springer, 500–517. https://doi.org/10.1007/3-540-45251-6_29

Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning invariants using decision trees and implication counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 499–512. https://doi.org/10.1145/2837614.2837664

Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. 2012. Synthesizing software verifiers from proof rules. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 405–416. https://doi.org/10.1145/2254064.2254112

Krystof Hoder and Nikolaj Bjørner. 2012. Generalized Property Directed Reachability. In *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings (Lecture Notes in Computer Science)*, Alessandro Cimatti and Roberto Sebastiani (Eds.), Vol. 7317. Springer, 157–171. https://doi.org/10.1007/978-3-642-31612-8_13

Hossein Hojjat and Philipp Rümmer. 2017. Deciding and Interpolating Algebraic Data Types by Reduction. In *19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2017, Timisoara, Romania, September 21-24, 2017*, Tudor Jebelean, Viorel Negru, Dana Petcu, Daniela Zaharie, Tetsuo Ida, and Stephen M. Watt (Eds.). IEEE Computer Society, 145–152. https://doi.org/10.1109/SYNASC.2017.00033

Hossein Hojjat and Philipp Rümmer. 2018. The ELDARICA Horn Solver. In *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, Nikolaj Bjørner and Arie Gurfinkel (Eds.). IEEE, 1–7. https://doi.org/10.23919/FMCAD.2018.8603013

Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2016. SMT-based model checking for recursive programs. *Formal Methods Syst. Des.* 48, 3 (2016), 175–205. https://doi.org/10.1007/s10703-016-0249-4

Yurii Kostyukov, Dmitry Mordvinov, and Grigory Fedyukovich. 2021. Beyond the elementary representations of program invariants over algebraic data types. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 451–465. https://doi.org/10.1145/3453483.3454055

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers (Lecture Notes in Computer Science)*, Edmund M. Clarke and Andrei Voronkov (Eds.), Vol. 6355. Springer, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

K. Rustan M. Leino. 2012. Automating Induction with an SMT Solver. In *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings (Lecture Notes in Computer Science)*, Viktor Kuncak and Andrey Rybalchenko (Eds.), Vol. 7148. Springer, 315–331. https://doi.org/10.1007/978-3-642-27940-9_21

Christof Löding, P. Madhusudan, and Lucas Peña. 2018. Foundations for natural proofs and quantifier instantiation. *Proc. ACM Program. Lang.* 2, POPL (2018), 10:1–10:30. https://doi.org/10.1145/3158098

Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2020. RustHorn: CHC-Based Verification for Rust Programs. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science)*, Peter Müller (Ed.), Vol. 12075. Springer, 484–514. https://doi.org/10.1007/978-3-030-44914-8_18

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings (Lecture Notes in Computer Science)*, Barbara Jobstmann and K. Rustan M. Leino (Eds.), Vol. 9583. Springer, 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

Derek C. Oppen. 1980. Reasoning About Recursively Defined Data Structures. *J. ACM* 27, 3 (1980), 403–411. https://doi.org/10.1145/322203.322204

Tuan-Hung Pham, Andrew Gacek, and Michael W. Whalen. 2016. Reasoning About Algebraic Data Types with Abstractions. *J. Autom. Reason.* 57, 4 (2016), 281–318. https://doi.org/10.1007/s10817-016-9368-2

Giles Reger and Andrei Voronkov. 2019. Induction in Saturation-Based Proof Search. In *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings (Lecture Notes in Computer Science)*, Pascal Fontaine (Ed.), Vol. 11716. Springer, 477–494. https://doi.org/10.1007/978-3-030-29436-6_28

Andrew Reynolds and Viktor Kuncak. 2015. Induction for SMT Solvers. In *Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings (Lecture Notes in Computer Science)*, Deepak D'Souza, Akash Lal, and Kim Guldstrand Larsen (Eds.), Vol. 8931. Springer, 80–98. https://doi.org/10.1007/978-3-662-46081-8_5

Dan Rosén and Nicholas Smallbone. 2015. TIP: Tools for Inductive Provers. In *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings (Lecture Notes in Computer Science)*, Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov (Eds.), Vol. 9450. Springer, 219–232. https://doi.org/10.1007/978-3-662-48899-7_16

Philippe Suter, Mirco Dotta, and Viktor Kuncak. 2010. Decision procedures for algebraic data types with abstractions. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 199–210. https://doi.org/10.1145/1706299.1706325

Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. 2011. Satisfiability Modulo Recursive Programs. In *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings (Lecture Notes in Computer Science)*, Eran Yahav (Ed.), Vol. 6887. Springer, 298–315. https://doi.org/10.1007/978-3-642-23702-7_23

Ting Zhang, Henny B. Sipma, and Zohar Manna. 2004. Decision Procedures for Recursive Data Structures with Integer Constraints. In *Automated Reasoning - Second International Joint Conference, IJCAR 2004, Cork, Ireland, July 4-8, 2004, Proceedings (Lecture Notes in Computer Science)*, David A. Basin and Michaël Rusinowitch (Eds.), Vol. 3097. Springer, 152–167. https://doi.org/10.1007/978-3-540-25984-8_9

Ting Zhang, Henny B. Sipma, and Zohar Manna. 2006. Decision procedures for term algebras with integer constraints. *Inf. Comput.* 204, 10 (2006), 1526–1574. https://doi.org/10.1016/j.ic.2006.03.004

He Zhu, Stephen Magill, and Suresh Jagannathan. 2018. A data-driven CHC solver. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 707–721. https://doi.org/10.1145/3192366.3192416