TPLP: Page 1–28.
doi:10.1017/S147106842400019X

Catamorphic Abstractions for Constrained Horn Clause Satisfiability[*]

EMANUELE DE ANGELIS
IASI-CNR, Rome, Italy
(email: emanuele.deangelis@iasi.cnr.it)

FABIO FIORAVANTI
DEc, University 'G. d'Annunzio', Chieti-Pescara, Italy
(email: fabio.fioravanti@unich.it)

ALBERTO PETTOROSSI
DICII, University of Rome 'Tor Vergata', Rome, Italy
(email: pettorossi@info.uniroma2.it)

MAURIZIO PROIETTI
IASI-CNR, Rome, Italy
(email: maurizio.proietti@iasi.cnr.it)

submitted 28 March 2024; accepted 16 August 2024

Abstract

Catamorphisms are functions that are recursively defined on list and trees and, in general, on algebraic data types (ADTs), and are often used to compute suitable abstractions of programs that manipulate ADTs. Examples of catamorphisms include functions that compute size of lists, orderedness of lists, and height of trees. It is well known that program properties specified through catamorphisms can be proved by showing the satisfiability of suitable sets of constrained Horn clauses (CHCs). We address the problem of checking the satisfiability of those sets of CHCs, and we propose a method for transforming sets of CHCs into equisatisfiable sets where catamorphisms are no longer present. As a consequence, clauses with catamorphisms can be handled without extending the satisfiability algorithms used by existing CHC solvers. Through an experimental evaluation on a nontrivial benchmark consisting of many list and tree processing algorithms expressed as sets of CHCs, we show that our technique is indeed effective and significantly enhances the performance of state-of-the-art CHC solvers.

KEYWORDS: program verification, constrained Horn clauses, catamorphisms, contracts.

[*] This is an extended version of the LOPSTR 2023 paper entitled: Constrained Horn Clauses Satisfiability via Catamorphic Abstractions, doi: https://doi.org/10.1007/978-3-031-45784-5_4

1 Introduction

Catamorphisms are functions that compute abstractions over algebraic data types (ADTs), such as lists or trees. The definition of a catamorphism is based on a simple recursion scheme, called a *fold* in the context of functional programming (Meijer et al., 1991). Examples of catamorphisms on lists of integers include functions that compute the orderedness of a list, the length of a list, and the sum of its elements. Similarly, examples of catamorphisms on trees are functions that compute the size of a tree, the height of a tree, and the minimum integer value at its nodes.

Through catamorphisms we can specify many useful program properties such as, for instance, the property that the list computed by a program for sorting lists is indeed sorted, or the property that the output list has the same length of the input list. For this reason, program analysis tools based on *abstract interpretation* (Cousot and Cousot, 1977; Hermenegildo et al., 2005) and *program verifiers* (Suter et al., 2011) have implemented special purpose techniques that handle catamorphisms.

In recent years, it has been shown that verification problems that use catamorphisms can be reduced to satisfiability problems for constrained Horn clauses (CHCs) by following a general approach that is very well suited for automatic proofs (Bjørner et al., 2015; De Angelis et al., 2022; Gurfinkel, 2022). A practical advantage of CHC-based verification is that it is supported by several CHC *solvers* which can be used as back-end tools (Blicha et al., 2022; De Angelis and Govind V. K., 2022; Komuravelli et al., 2016; Hojjat and Rümmer, 2018).

Unfortunately, the direct translation of catamorphism-based verification problems into CHCs is not always helpful, because CHC solvers often lack mechanisms for computing solutions by performing induction over ADTs. To overcome this difficulty, some CHC solvers have been extended with special purpose satisfiability algorithms that handle (some classes of) catamorphisms (Govind et al., 2022; Hojjat and Rümmer, 2018; Kostyukov et al., 2021; Gurfinkel, 2022). For instance, the module of Eldarica for solving CHCs has been extended by allowing constraints that use the built-in *size* function counting the number of function symbols in the ADTs (Hojjat and Rümmer, 2018).

In this paper, we consider a class of catamorphisms that is strictly larger than the ones handled by the above mentioned satisfiability algorithms, and we follow an approach based on the transformation of CHCs (De Angelis et al., 2022, 2023). In particular, given a set P of CHCs that uses catamorphisms and includes one or more queries encoding the properties of interest, we transform P into a new set P' such that: (i) P is satisfiable if and only if P' is satisfiable, and (ii) no catamorphism is present in P'. Thus, the satisfiability of P' can be verified by a CHC solver that is not extended for handling catamorphisms.

The main difference between the technique we present in this paper and the above cited works (De Angelis et al., 2022, 2023) is that the algorithm we present here does not require that we specify suitable properties of how the catamorphisms relate to every predicate occurring in the given set P of CHCs. For instance, if we want to verify that the output list S of the set of CHCs defining $quicksort(L, S)$ has the same length of the input list L, we need not specify that, for the auxiliary predicate $partition(X, Xs, Ys, Zs)$ that divides the list Xs into the two lists Ys and Zs, it is the case that the length of Xs is the sum of the lengths of Ys and Zs. This property can automatically be derived by the

CHC solver when it looks for a model of the set of CHCs obtained by transformation. In this sense, our technique may allow the discovery of some lemmas needed for the proof of the property of interest.

We will show through a benchmark set of list and tree processing algorithms expressed as sets of CHCs, that our transformation technique is indeed effective and is able to drastically increase the performance of state-of-the-art CHC solvers such as Eldarica (Hojjat and Rümmer, 2018) (with the built-in catamorphism *size*) and Z3 with the SPACER engine (de Moura and Bjørner, 2008; Komuravelli et al., 2016).

The rest of the paper is organized as follows. In Section 2, we recall some preliminary notions on CHCs and catamorphisms. In Section 3 we show an introductory example to motivate our technique. In Section 4 we present our transformation algorithm and prove that it guarantees the equisatisfiability of the initial sets of CHCs and the transformed sets of CHCs. In Section 5 we present the implementation of our technique in the VeriCaT$_{abs}$ tool, and through an experimental evaluation, we show the beneficial effect of the transformation on both Eldarica and Z3 CHC solvers. We will consider several abstractions based on catamorphisms relative to lists and trees, such as size, minimum element, orderedness, element membership, element multiplicity, and combinations thereof. Finally, in Section 6, we discuss related work and we outline future research directions.

2 Basic notions

The programs and the properties we consider in this paper are expressed as sets of constrained Horn clauses written in a many-sorted first-order language $\mathcal{L}$ with equality (=). Constraints are expressions of the linear integer arithmetic (*LIA*) and the boolean algebra (*Bool*). The theories of *LIA* and *Bool* will be collectively denoted by $LIA \cup Bool$. The equality symbol = will be used both for integers and booleans. In particular, a *constraint* is a quantifier-free formula c, where *LIA* constraints may occur as subexpressions of boolean constraints, according to the SMT approach (Barrett et al., 2009). The syntaxes of a constraint c and an elementary *LIA* constraint d are as follows:

$$\begin{aligned} c &::= d \mid B \mid \mathit{true} \mid \mathit{false} \mid {\sim}c \mid c_1 \,\&\, c_2 \mid c_1 \vee c_2 \mid c_1 \Rightarrow c_2 \mid c_1 = c_2 \mid \\ &\qquad \mathit{ite}(c, c_1, c_2) \mid t = \mathit{ite}(c, t_1, t_2) \\ d &::= t_1 < t_2 \mid t_1 \leq t_2 \mid t_1 = t_2 \mid t_1 \geq t_2 \mid t_1 > t_2 \end{aligned}$$

where:(i) B is a boolean variable, (ii) $\sim$, &, $\vee$, and $\Rightarrow$ denote negation, conjunction, disjunction, and implication, respectively, (iii) the ternary function *ite* denotes the if-then-else operator (i.e. $\mathit{ite}(c, c_1, c_2)$ has the following semantics: if c then c_1 else c_2), and (iv) t, possibly with subscripts, t, t_1 and t_2 is a *LIA* term of the form $a_0 + a_1X_1 + \ldots + a_nX_n$ with integer coefficients $a_0, \ldots, a_n$ and integer variables $X_1, \ldots, X_n$.

The integer and boolean sorts are said to be *basic sorts*. A recursively defined sort (such as the sort of lists and trees) is said to be an *algebraic data type* (ADT, for short).

An *atom* is a formula of the form $p(t_1, \ldots, t_m)$, where p is a predicate symbol not occurring in $LIA \cup Bool$, and $t_1, \ldots, t_m$ are first-order terms in $\mathcal{L}$. A *constrained Horn clause* (CHC), or simply, a *clause*, is an implication of the form $H \leftarrow c, G$. The conclusion

H, called the *head*, is either an atom or *false*, and the premise, called the *body*, is the conjunction of a constraint c and a conjunction G of zero or more atoms. G is said to be a *goal*. A clause is said to be a *query* if its head is *false*, and a *definite clause*, otherwise. Without loss of generality, at the expense of introducing suitable equalities, we assume that every atom of the body of a clause has distinct variables (of any sort) as arguments. Given an expression e, by $vars(e)$ we denote the set of all variables occurring in e. By $bvars(e)$ (or $adt\text{-}vars(e)$) we denote the set of variables in e whose sort is a basic sort (or an ADT sort, respectively). The *universal closure* of a formula φ is denoted by $\forall(\varphi)$.

A $\mathbb{D}$-interpretation for a set S of CHCs is an interpretation where the symbols of $LIA \cup Bool$ are interpreted as usual. A $\mathbb{D}$-interpretation I is said to be a $\mathbb{D}$-model of S if all clauses of S are true in I. A set S of CHCs is said to be $\mathbb{D}$-*satisfiable* (or *satisfiable*, for short) if it has a $\mathbb{D}$-model, and it is said to be $\mathbb{D}$-*unsatisfiable* (or *unsatisfiable*, for short), otherwise.

Given a set P of definite clauses, there exists a *least* $\mathbb{D}$-model of P, denoted $M(P)$ (Jaffar and Maher, 1994). Let P be a set of definite clauses and for $i=1,\ldots,n$, Q_i be a query. Then $P \cup \{Q_1,\ldots,Q_n\}$ is satisfiable if and only if, for $i=1,\ldots,n$, $M(P) \models Q_i$.

The catamorphisms we consider in this paper are defined by first-order, relational recursive schemata as we now indicate. Similar definitions are introduced also in (higher-order) functional programming (Meijer et al., 1991; Hinze et al., 2013).

Let f be a predicate symbol with $m+n$ arguments (for $m \geq 0$ and $n \geq 0$) with sorts $\alpha_1,\ldots,\alpha_m$, $\beta_1,\ldots,\beta_n$, respectively. We say that f is a *functional predicate* from sort $\alpha_1 \times \ldots \times \alpha_m$ to sort $\beta_1 \times \ldots \times \beta_n$, with respect to a given set P of definite clauses that define f, if $M(P) \models \forall X,Y,Z.\ f(X,Y) \wedge f(X,Z) \rightarrow Y=Z$, where X is an m-tuple of distinct variables, and Y and Z are n-tuples of distinct variables. In this case, when we write the atom $f(X,Y)$, we mean that X and Y are the tuples of the *input* and *output* variables of f, respectively. We say that f is a *total predicate* if $M(P) \models \forall X \exists Y.\ f(X,Y)$. In what follows, a 'total, functional predicate' f from a tuple α of sorts to a tuple β of sorts is said to be a 'total function' in $[\alpha \rightarrow \beta]$, and it is denoted by $f \in [\alpha \rightarrow \beta]$.

Now we introduce the notions of a list catamorphism and a binary tree catamorphism. We leave to the reader the task of introducing, the definitions of similar catamorphisms for recursively defined algebraic data types that may be needed for expressing the properties of interest. Let α, β, γ, and δ be (products of) basic sorts. Let $list(\beta)$ be the sort of lists with elements of sort β, and $btree(\beta)$ be the sort of binary trees with values of sort β.

Definition 1 (List and Binary Tree Catamorphisms).
A *list catamorphism* ℓ is a total function in $[\alpha \times list(\beta) \rightarrow \gamma]$ defined as follows:

L1. $\ell(X,[\,],Y) \leftarrow \ell\_basis(X,Y)$
L2. $\ell(X,[H|T],Y) \leftarrow f(X,T,Rf),\ \ell(X,T,R),\ \ell\_combine(X,H,R,Rf,Y)$

where: (i) $\ell\_basis \in [\alpha \rightarrow \gamma]$, (ii) $\ell\_combine \in [\alpha \times \beta \times \gamma \times \delta \rightarrow \gamma]$, and (iii) f is itself a list catamorphism in $[\alpha \times list(\beta) \rightarrow \delta]$.
A *binary tree catamorphism* bt is a total function in $[\alpha \times btree(\beta) \rightarrow \gamma]$ defined as follows:

BT1. $bt(X, leaf, Y) \leftarrow bt\_basis(X,Y)$
BT2. $bt(X, node(L,N,R), Y) \leftarrow g(X,L,RLg),\ g(X,R,RRg),$
$\quad bt(X,L,RL),\ bt(X,R,RR),\ bt\_combine(X,N,RL,RR,RLg,RRg,Y)$

https://doi.org/10.1017/S147106842400019X Published online by Cambridge University Press

*** Initial set of CHCs including the catamorphism *listcount*.

1. $double(Xs,Zs) \leftarrow eq(Xs,Ys),\; append(Xs,Ys,Zs)$
2. $eq(Xs,Xs) \leftarrow$
3. $append([\,],Ys,Ys) \leftarrow$
4. $append([X|Xs],Ys,[X|Zs]) \leftarrow append(Xs,Ys,Zs)$
5. $listcount(X,[\,],N) \leftarrow N=0$
6. $listcount(X,[H|T],N) \leftarrow N=ite(X{=}H, NT{+}1, NT),\; listcount(X,T,NT)$

*** Query.

7. $false \leftarrow M=2\,N{+}1,\; listcount(X,Zs,M),\; double(Xs,Zs)$

Fig. 1. The initial set of CHCs (clauses 1–6) and query 7 that specifies that the number of occurrences of an element X in the list Zs is even.

where : (i) $bt\_basis \in [\alpha \to \gamma]$, (ii) $bt\_combine \in [\alpha\times\beta\times\gamma\times\gamma\times\delta\times\delta \to \gamma]$, and (iii) g is itself a binary tree catamorphism in $[\alpha\times btree(\beta) \to \delta]$.

Instances of the schemas of the list catamorphisms and the binary tree catamorphisms (see Definition 1 above) may lack some components, such as the parameter X of basic sort α, or the catamorphisms f or g. The possible presence of these components makes the class of catamorphisms considered in this paper strictly larger than the ones used by other CHC-based approaches (Govind V. K. et al., 2022; Hojjat and Rümmer, 2018; Kostyukov et al., 2021; Gurfinkel, 2022).

3 An introductory example

Let us consider a set of CHCs for doubling lists of integers (see clauses 1–4 in Figure 1). We have that: (i) $double(Xs,Zs)$ holds if and only if list Zs is the concatenation of two copies of the same list Xs of integers, (ii) $eq(Xs,Ys)$ holds if and only if list Xs is equal to list Ys, and (iii) $append(Xs,Ys,Zs)$ holds if and only if list Zs is the result of concatenating list Ys to the right of list Xs.

Let us assume that we want to verify the following *Even* property: if $double(Xs,Zs)$ holds, then for any integer X, the number of occurrences of X in Zs is an even number. In order to do so, we use the list catamorphism $listcount(X,Zs,M)$ (see clauses 5–6 in Figure 1) that holds if and only if M is the number of occurrences of X in list Zs. Note that $listcount(X,Zs,M)$ is indeed a list catamorphism because clauses 5–6 are instances of clauses $L1$–$L2$ in Definition 1, when: (i) ℓ is *listcount*, (ii) Y is N, (iii) $\ell\_basis(X,Y)$ is the *LIA* constraint $N{=}0$, (iv) $f(X,T,Rf)$ is absent, and (v) $\ell\_combine(X,H,R,Rf,Y)$ is the *LIA* constraint $N{=}\,ite(X{=}H, NT{+}1, NT)$.

Our verification task can be expressed as query 7 in Figure 1, whereby we derive *false* if the number M of occurrences of X in Zs is odd (recall that we assume that $M{=}2\,N{+}1$ is a *LIA* constraint).

Now, neither the CHC solver Eldarica nor Z3 is able to prove the satisfiability of clauses 1–7 and thus, those solvers are not able to show the *Even* property. By the transformation technique we will propose in this paper, we get a new set of clauses whose satisfiability can be shown by Z3 and thus, the *Even* property is proved.

To perform this transformation, we use the information that the property to be verified is expressed through the catamorphism *listcount*. However, in contrast to previous approaches (De Angelis et al., 2022, 2023), we need *not* specify any property of the catamorphism *listcount* when it acts upon the predicates *double*, *eq*, and *append*. For instance, we need not specify that if Zs is the concatenation of Xs and Ys, then for any X, the number of occurrences of X in Zs is the sum of the numbers of occurrences of X in Xs and Ys. Indeed, in the approach we propose in this paper, we have only to specify the association of every ADT sort with a suitable catamorphism or, in general, a conjunction of catamorphisms. In particular, in our *double* example, we associate the sort of integer lists, denoted $list(int)$, with the catamorphism *listcount*. Then, we rely on the CHC solver for the discovery, after the transformation described in the following sections, of suitable relations between the variables that represent the output of the *listcount* catamorphism atoms. Thus, by applying the technique proposed in this paper, much less ingenuity is required on the part of the programmer for verifying program correctness with respect to the previously proposed approaches.

Our transformation technique introduces, for each predicate p occurring in the initial set of CHCs, a new predicate *newp* defined by the conjunction of a p atom and, for each argument of p with ADT sort τ, the catamorphism atom(s) with which τ has been associated. In particular, in the case of our *double* example, for the predicate *double* we introduce the new predicate $new1$ (for simplicity, we call it $new1$, instead of *newdouble*) whose definition is clause $D1$ in Figure 2. The body of that clause is the conjunction of the atom $double(B, E)$ and two *listcount* catamorphism atoms, one for each of the integer lists B and E, as *listcount* is the catamorphism with which the sort of integer lists has been associated. Similarly, for the predicates *append* and *eq* whose definitions are respectively clauses $D2$ and $D3$ listed in Figure 2.

Thus, we derive a new version of the initial CHCs where each predicate p has been replaced by the corresponding *newp*. Then, by applying variants of the fold/unfold transformation rules, we derive a final, transformed set of CHCs. When the CHC solver looks for a model of this final set of CHCs, it is guided by the fact that suitable constraints, inferred from the query, must hold among the arguments of the newly introduced predicates, such as *newp*, and thus, the solver can often be more effective.

In our transformation we also introduce, for each predicate *newp*, a predicate called *newp_woADTs* whose definition is obtained by removing the ADT arguments from the definition of *newp*. For the CHC solvers, it is often easier to find a model for *newp_woADTs*, rather than for *newp*, because the solvers need not handle ADTs at all. However, since each *newp_woADTs* is an overapproximation of *newp*, by using the clauses with the ADTs removed, one could wrongly infer unsatisfiability in cases when, on the contrary, the initial set of CHCs is satisfiable.

Now, in order to make it easier for the solvers to show satisfiability of sets of CHCs and, at the same time, to guarantee the equisatisfiability of the derived set of clauses with respect to the initial set, we add to every atom in the body of every derived clause for *newp* the corresponding atom without ADT arguments (see Theorem 1 for the correctness of these atom additions). By performing these transformation steps starting from clauses 1–6 and query 7 (listed in Figure 1) together with the specification that every variable of sort $list(int)$ should be associated with a *listcount* atom, we derive using our

*** The new predicate definitions introduced during transformation. In these definitions the underlined variables have an ADT sort (i.e., they are integer lists), while the non-underlined variables have a basic sort (i.e., they are integer numbers).

$D1.\ new1(A,\underline{B},C,\underline{E},F) \leftarrow listcount(A,\underline{B},C),\ listcount(A,\underline{E},F),\ double(\underline{B},\underline{E})$
$D2.\ new2(A,\underline{B},C,\underline{E},F,\underline{H},I) \leftarrow listcount(A,\underline{B},C),\ listcount(A,\underline{E},F),\ listcount(A,\underline{H},I),$
$\quad append(\underline{H},\underline{B},\underline{E})$
$D3.\ new3(A,\underline{B},C,\underline{E},F) \leftarrow listcount(A,\underline{B},C),\ listcount(A,\underline{E},F),\ eq(\underline{E},\underline{B})$
$D4.\ new4(A,\underline{B},C) \leftarrow listcount(A,\underline{B},C)$

*** Clauses derived after transformation from clauses 1–7 of Figure 1.

11. $false \leftarrow C=2\,D+1,\ new1(A,E,F,G,C),\ new1\_woADTs(A,F,C)$
12. $new1(A,B,C,E,F) \leftarrow new2(A,M,K,E,F,B,C),\ new2\_woADTs(A,K,F,C),$
$new3(A,M,K,B,C),\ new3\_woADTs(A,K,C)$
13. $new2(A,B,C,B,C,[\,],G) \leftarrow G=0,\ new4(A,B,C),\ new4\_woADTs(A,C)$
14. $new2(A,B,C,[E|F],G,[E|J],K) \leftarrow G=ite(A=E,N+1,N),\ K=ite(A=E,P+1,P),$
$new2(A,B,C,F,N,J,P),\ new2\_woADTs(A,C,N,P)$
15. $new3(A,B,C,B,C) \leftarrow new4(A,B,C),\ new4\_woADTs(A,C)$
16. $new4(A,[\,],B) \leftarrow B=0$
17. $new4(A,[B|C],D) \leftarrow D=ite(A=B,F+1,F),\ new4(A,C,F),\ new4\_woADTs(A,F)$

18. $new1\_woADTs(A,B,D) \leftarrow new2\_woADTs(A,I,D,B),\ new3\_woADTs(A,I,B)$
19. $new2\_woADTs(A,B,B,F) \leftarrow F=0,\ new4\_woADTs(A,B)$
20. $new2\_woADTs(A,B,D,F) \leftarrow D=ite(A=I,K+1,K),\ F=ite(A=I,L+1,L),$
$new2\_woADTs(A,B,K,L)$
21. $new3\_woADTs(A,B,B) \leftarrow new4\_woADTs(A,B)$
22. $new4\_woADTs(A,B) \leftarrow B=0.$
23. $new4\_woADTs(A,B) \leftarrow B=ite(A=C,D+1,D),\ new4\_woADTs(A,D).$

Fig. 2. Clauses $D1$–$D4$ are the predicate definitions introduced during transformation. Clauses 11–23 are the clauses derived after transformation from clauses 1–6 and query 7 of Figure 1.

transformation algorithm $\mathcal{T}_{abs}$ (see Section 4) clauses 11–23 listed in Figure 2. These derived clauses are indeed shown to be satisfiable by the Z3 solver, and thus the *Even* property is proved.

4 CHC transformation via catamorphic abstractions

In this section we present our transformation algorithm, called $\mathcal{T}_{abs}$, whose input is: (i) a set P of definite clauses, (ii) a query Q expressing the property to be verified, and (iii) for each ADT sort, a conjunction of catamorphisms whose definitions are included in P. Algorithm $\mathcal{T}_{abs}$ introduces a set of new predicates, which incorporate as extra arguments some information coming from the catamorphisms, and transforms $P \cup \{Q\}$ into a new set $P' \cup \{Q'\}$ such that $P \cup \{Q\}$ is satisfiable if and only if so is $P' \cup \{Q'\}$.

The transformation is effective when the catamorphisms used in the new predicate definitions establish relations that are useful to solve the query. In particular, it is often helpful to use in the new definitions catamorphisms that include the ones occurring in the query, such as the catamorphism *listcount* of our introductory *double* example. However, as we will see later, there are cases in which it is important to consider catamorphisms not present in the query (see Example 2). The choice of the suitable catamorphisms to be used in the transformation rests upon the programmer's ingenuity and on her/his understanding of the program behavior. The problem of chosing the most suitable catamorphisms in a fully automatic way is left for future research.

4.1 Catamorphic abstraction specifications

The predicates in P different from catamorphisms are called *program predicates*. An atom whose predicate is a program predicate is called a *program atom* and an atom whose predicate is a catamorphism predicate is called a *catamorphism atom*. Without loss of generality, we assume that no clause in P has occurrences of both program atoms and catamorphism atoms. The query Q given in input to $\mathcal{T}_{abs}$ is of the form:

$$\mathit{false} \leftarrow c, \mathit{cata}_1(X, T_1, Y_1), \ldots, \mathit{cata}_n(X, T_n, Y_n), p(Z)$$

where: (i) $p(Z)$ is a program atom and Z is a tuple of distinct variables; (ii) cata_1, $\ldots, \mathit{cata}_n$ are catamorphism predicates; (iii) c is a constraint; (iv) X is a tuple of distinct variables of basic sort; (v) $T_1, \ldots, T_n$ are ADT variables occurring in Z; and (vi) $Y_1, \ldots, Y_n$ are pairwise disjoint tuples of distinct variables of basic sort not occurring in $\mathit{vars}(\{X, Z\})$. Without loss of generality, we assume that the cata_i's over the same ADT variable are all distinct (this assumption is trivially satisfied by query 7 of Figure 1). For each ADT sort τ, a *catamorphic abstraction for* τ is a conjunction of catamorphisms defined as follows:

$$\mathit{cata}_\tau(X, T, Y_1, \ldots, Y_n) =_{def} \mathit{cata}_1(X, T, Y_1), \ldots, \mathit{cata}_n(X, T, Y_n)$$

where: (i) T is a variable of ADT sort τ, (ii) $X, Y_1, \ldots, Y_n$ are tuples of variables of basic sort, (iii) the variables in $\{X, Y_1, \ldots, Y_n\}$ are all distinct, and (iv) the cata_i predicates are all distinct.

Given catamorphic abstractions for the ADT sorts $\tau_1, \ldots, \tau_k$, a *catamorphic abstraction specification for* the set P of CHCs is a set of expressions, one expression for each program predicate p in P that has at least one argument of ADT sort. The expression for the predicate p is called the *catamorphic abstraction specification for* p and it is of the form:

$$p(Z) \Longrightarrow \mathit{cata}_{\tau_1}(X, T_1, V_1), \ldots, \mathit{cata}_{\tau_k}(X, T_k, V_k)$$

where: (i) Z is a tuple of distinct variables, (ii) $T_1, \ldots, T_k$ are the distinct variables in Z of (not necessarily distinct) ADT sorts $\tau_1, \ldots, \tau_k$, respectively, (iii) $V_1, \ldots, V_k$ are pairwise disjoint tuples of distinct variables of basic sort not occurring in $\mathit{vars}(\{X, Z\})$; and (iv) $\mathit{vars}(X) \cap \mathit{vars}(Z) = \emptyset$.

Example 1.
Let us consider our introductory *double* example (see Figure 1) and the catamorphic abstraction for the sort $\mathit{list}(\mathit{int})$:

$$\mathit{cata}_{\mathit{list}(\mathit{int})}(X, L, N) =_{def} \mathit{listcount}(X, L, N)$$

https://doi.org/10.1017/S147106842400019X Published online by Cambridge University Press

This abstraction determines the following catamorphic abstraction specifications for the predicates *double*, *eq*, and *append* (and thus, for the set $\{1, \ldots, 6\}$ of clauses):

$$\begin{aligned}
&\mathit{double}(Xs, Zs) \Longrightarrow \mathit{listcount}(X, Xs, N1),\ \mathit{listcount}(X, Zs, N2)\\
&\mathit{eq}(Xs, Zs) \Longrightarrow \mathit{listcount}(X, Xs, N1),\ \mathit{listcount}(X, Zs, N2)\\
&\mathit{append}(Xs,Ys,Zs) \Longrightarrow \mathit{listcount}(X,Xs,N1), \mathit{listcount}(X,Ys,N2), \mathit{listcount}(X,Zs,N3)
\end{aligned}$$

Note that no relationships among the variables $N1$, $N2$, and $N3$ are stated by the specifications. Those relationships will be discovered by the solver after transformation. □

Example 2.
Let us consider: (i) a set *Quicksort* of clauses where predicate $\mathit{quicksort}(L, S)$ holds if S is obtained from list L by the quicksort algorithm and (ii) the following query:

$$\mathit{false} \leftarrow BS{=}\mathit{false},\ \mathit{is\_asorted}(S,BS),\ \mathit{quicksort}(L, S) \qquad (\mathit{Ord})$$

where $\mathit{is\_asorted}(S,BS)$ returns $BS{=}\mathit{true}$ if the elements of S are ordered in weakly ascending order, and $BS{=}\mathit{false}$, otherwise. The catamorphism *is_asorted* is defined in term of the catamorphism *hd*, as follows:

$$\begin{aligned}
&\mathit{is_asorted}([\,], B) \leftarrow B{=}\mathit{true}\\
&\mathit{is_asorted}([H|T], B) \leftarrow B{=}(\mathit{IsDef} \Rightarrow (H{\le}\mathit{HdT}\ \&\ BT)),\\
&\qquad\qquad\qquad \mathit{hd}(T, \mathit{IsDef}, \mathit{HdT}),\ \mathit{is_asorted}(T, BT)\\
&\mathit{hd}([\,], \mathit{IsDef}, \mathit{Hd}) \leftarrow\ \mathit{IsDef}{=}\mathit{false},\ \mathit{Hd}{=}0\\
&\mathit{hd}([H|T], \mathit{IsDef}, \mathit{Hd}) \leftarrow \mathit{IsDef}{=}\mathit{true},\ \mathit{Hd}{=}H.
\end{aligned}$$

$\mathit{hd}(L, \mathit{IsDef}, \mathit{Hd})$ holds if *either* L is the empty list ($\mathit{IsDef}{=}\mathit{false}$) and Hd is 0 or L is a nonempty list ($\mathit{IsDef}{=}\mathit{true}$) and Hd is its head. Thus, *hd* is a total function. Note that the arbitrary value 0 is not used in the clauses for *is_asorted*.

Let us consider a catamorphic abstraction $\mathit{cata}_{\mathit{list}(\mathit{int})}$ for the sort $\mathit{list}(\mathit{int})$, which is the sort of the variables L and S in $\mathit{quicksort}(L, S)$. That abstraction, consisting of the conjunction of three list catamorphisms *listmin*, *listmax*, and *is_asorted*, is defined as follows:

$$\begin{aligned}
&\mathit{cata}_{\mathit{list}(\mathit{int})}(L, \mathit{BMinL}, \mathit{MinL}, \mathit{BMaxL}, \mathit{MaxL}, \mathit{BL}) =_{\mathit{def}}\\
&\qquad \mathit{listmin}(L, \mathit{BMinL}, \mathit{MinL}),\ \mathit{listmax}(L, \mathit{BMaxL}, \mathit{MaxL}),\ \mathit{is_asorted}(L, \mathit{BL})
\end{aligned}$$

where: (i) if L is not empty, $\mathit{listmin}(L, \mathit{BMinL}, \mathit{MinL})$ holds if $\mathit{BMinL}{=}\mathit{true}$ and MinL is the minimum integer in L, and (ii) otherwise, if L is empty, $\mathit{listmin}(L, \mathit{BMinL}, \mathit{MinL})$ holds if $\mathit{BMinL}{=}\mathit{false}$ and $\mathit{MinL}{=}0$. If $\mathit{BMinL}{=}\mathit{false}$, then MinL should not be used elsewhere in the clause where $\mathit{listmin}(L, \mathit{BMinL}, \mathit{MinL})$ occurs. Analogously for *listmax*, instead of *listmin*. Then, the catamorphic abstraction specification for *quicksort* is as follows:

$$\begin{aligned}
&\mathit{quicksort}(L, S) \Longrightarrow\\
&\quad \mathit{listmin}(L,\mathit{BMinL},\mathit{MinL}),\ \mathit{listmax}(L,\mathit{BMaxL},\mathit{MaxL}),\ \mathit{is_asorted}(L,\mathit{BL}),\\
&\quad \mathit{listmin}(S,\mathit{BMinS},\mathit{MinS}),\ \mathit{listmax}(S,\mathit{BMaxS},\mathit{MaxS}),\ \mathit{is_asorted}(S,\mathit{BS})
\end{aligned}$$

Now, let us assume that in the set of clauses defining $\mathit{quicksort}(L, S)$, we have the atom $\mathit{partition}(V,L,A,B)$ that, given the integer V and the list L, holds if A is the list made

https://doi.org/10.1017/S147106842400019X Published online by Cambridge University Press

out of the elements of L not larger than V, and B is the list made out of the remaining elements of L larger than V. We have that the catamorphic abstraction specification for *partition* which has the list arguments L, A, and B, is as follows:

$$\begin{array}{l} \mathit{partition}(V,L,A,B) \Longrightarrow \\ \quad \mathit{listmin}(L,\mathit{BMinL},\mathit{MinL}),\ \mathit{listmax}(L,\mathit{BMaxL},\mathit{MaxL}),\ \mathit{is\_asorted}(L,\mathit{BL}), \\ \quad \mathit{listmin}(A,\mathit{BMinA},\mathit{MinA}),\ \mathit{listmax}(A,\mathit{BMaxA},\mathit{MaxA}),\ \mathit{is\_asorted}(A,\mathit{BA}), \\ \quad \mathit{listmin}(B,\mathit{BMinB},\mathit{MinB}),\ \mathit{listmax}(B,\mathit{BMaxB},\mathit{MaxB}),\ \mathit{is\_asorted}(B,\mathit{BB}) \end{array}$$

Note that the catamorphisms *listmin* and *listmax* are not present in the query *Ord*. However, they are needed for stating the property that, if $\mathit{partition}(V,L,A,B)$ holds, then the maximum element of the list A is less than or equal to the minimum element of the list B. This is a key property useful for proving the orderedness of the list S constructed by $\mathit{quicksort}(L, S)$. The fact that the catamorphisms *listmin* and *listmax* are helpful in the proof of the orderedness of S rests upon programmer's intuition. However, in our approach the programmer need not explicitly state all the properties of *listmin* and *listmax* which are needed for the proof. Indeed, the relationships among the output variables of *listmin* and *listmax* are automatically inferred by the CHC solver. □

4.2 Transformation rules

The rules for transforming CHCs that use catamorphisms are variants of the usual fold/unfold rules for CHCs (De Angelis et al., 2022).

A *transformation sequence* from an initial set S_0 of CHCs to a final set S_n of CHCs is a sequence $S_0 \Rrightarrow S_1 \Rrightarrow \ldots \Rrightarrow S_n$ of sets of CHCs such that, for $i{=}0, \ldots, n{-}1$, S_{i+1} is derived from S_i, denoted $S_i \Rrightarrow S_{i+1}$, by performing a transformation step consisting in applying one of the following transformation Rules R1–R5.

The objective of a transformation sequence constructed by algorithm $\mathcal{T}_{abs}$ is to derive from a given set S_0 a new, equisatisfiable set S_n in which for each program predicate p in S_0, there is a new predicate *newp* whose definition is given by the conjunction of an atom for p with some catamorphism atoms. With respect to p, the predicate *newp* has extra arguments that hold the values of the catamorphisms for the arguments of p with ADT sort.

(R1) *Definition Rule.* Let D be a clause of the form $\mathit{newp}(X_1, \ldots, X_k) \leftarrow \mathit{Catas}, A$, where: (1) *newp* is a predicate symbol not occurring in the sequence $S_0 \Rrightarrow S_1 \Rrightarrow \ldots \Rrightarrow S_i$ constructed so far, (2) $\{X_1, \ldots, X_k\} = \mathit{vars}(\{\mathit{Catas}, A\})$, (3) *Catas* is a conjunction of catamorphism atoms, with $\mathit{adt\text{-}vars}(\mathit{Catas}) \subseteq \mathit{adt\text{-}vars}(A)$, and (4) A is a program atom. By the *definition introduction rule* we add D to S_i and we get the new set $S_{i+1} = S_i \cup \{D\}$.

We will say that D is a definition *for* A.

For any $i{\geq}0$, by Defs_i we denote the set of clauses, called *definitions*, introduced by Rule R1 during the construction of the sequence $S_0 \Rrightarrow S_1 \Rrightarrow \ldots \Rrightarrow S_i$.

Example 3.

In our *double* example, by applying the definition rule we may introduce the following clause, whose variables of sort $\mathit{list}(\mathit{int})$ are B and E (the underlining of the list variables B and E has been omitted here):

https://doi.org/10.1017/S147106842400019X Published online by Cambridge University Press

$D1.\ new1(A,B,C,E,F) \leftarrow listcount(A,B,C),\ listcount(A,E,F),\ double(B,E)$

Thus, $S_1 = S_0 \cup \{D1\}$, where S_0 consists of clauses 1–7 of Figure 1. □

By making use of the *Unf* function (see Definition 2), we introduce the unfolding rule (see Rule R2), which consists of some unfolding steps followed by the application of the functionality property, which was presented in previous work (De Angelis et al., 2022). Recall that list and binary tree catamorphisms and, in general, all catamorphisms are assumed to be total functions (see Definition 1).

Definition 2 (One-step Unfolding).
Let D: $H \leftarrow c, L, A, R$ be a clause, where A is an atom, and let P be a set of definite clauses with $vars(D) \cap vars(P) = \emptyset$. Let $K_1 \leftarrow c_1, B_1, \ldots, K_m \leftarrow c_m, B_m$, with $m \geq 0$, be the clauses in P, such that, for $j = 1, \ldots, m$: (i) there exists a most general unifier ϑ_j of A and K_j, and (ii) the conjunction of constraints $(c, c_j)\vartheta_j$ is satisfiable.
One-step unfolding produces the following set of CHCs:

$$Unf(D, A, P) = \{(H \leftarrow c, c_j, L, B_j, R)\vartheta_j \mid j = 1, \ldots, m\}.$$

In the sequel, *Catas* denotes a conjunction of catamorphism atoms.

(R2) *Unfolding Rule.* Let D: $newp(U) \leftarrow Catas, A$ be a definition in $S_i \cap Defs_i$, where A is a program atom, and P be the set of definite clauses in S_i. We derive a new set *UnfD* of clauses by the following three steps.
Step 1. (*One-step unfolding of program atom*) $UnfD := Unf(D, A, P)$;
Step 2. (*Unfolding of the catamorphism atoms*)
while there exists a clause E: $H \leftarrow d, L, C, R$ in *UnfD*, for some conjunctions L and R of atoms, such that C is a catamorphism atom whose argument of ADT sort is not a variable **do**

$UnfD := (UnfD \setminus \{E\}) \cup Unf(E, C, P)$;

Step 3. (*Applying Functionality on catamorphism atoms*)
while there exists a clause E: $H \leftarrow d, L, cata(X, T, Y1), cata(X, T, Y2), R$ in *UnfD*, for some catamorphism *cata* **do**

$UnfD := (UnfD - \{E\}) \cup \{H \leftarrow d, Y1{=}Y2, L, cata(X, T, Y1), R\}$.

Then, by *unfolding* clause D, we get the new set of clauses $S_{i+1} = (S_i \setminus \{D\}) \cup UnfD$.

Example 4.
For instance, in our *double* example, by unfolding clause $D1$ we get:

$E1.\ new1(A,B,C,E,F) \leftarrow listcount(A,B,C),\ listcount(A,E,F),\ eq(B,G),\ append(B,G,E)$

Thus, $S_2 = S_0 \cup \{E1\}$. □

By the following *catamorphism addition rule*, we use the catamorphic abstraction specifications for adding catamorphism atoms to the bodies of clauses. Here and in what follows, for any two conjunctions G_1 and G_2 of atoms, we say that G_1 is a *subconjunction* of G_2 if every atom of G_1 is an atom of G_2.

(R3) *Catamorphism Addition Rule.* Let C: $H \leftarrow c, Catas, A_1, \ldots, A_m$ be a clause in S_i, where H is either *false* or a program atom, and $A_1, \ldots, A_m$ are program atoms. Let E be the clause derived from C as follows:

https://doi.org/10.1017/S147106842400019X Published online by Cambridge University Press

for $k=1,\ldots,m$ **do**

- let Catas_k be the conjunction of every catamorphism atom F in Catas such that $\mathit{adt\text{-}vars}(A_k)\cap \mathit{adt\text{-}vars}(F)\neq\emptyset$;
- let $A_k \Longrightarrow \mathit{cata}_1(X,T_1,Y_1),\ldots,\mathit{cata}_n(X,T_n,Y_n)$ be a catamorphic abstraction specification for the predicate of A_k, where the variables in $Y_1,\ldots,Y_n$ do not occur in C, and the conjunction $\mathit{cata}_1(X,T_1,Y_1),\ldots,\ \mathit{cata}_n(X,T_n,Y_n)$ can be split into two subconjunctions B_1 and B_2 such that:
  - (i) a variant $B_1\vartheta$ of B_1, for a substitution ϑ acting on $\mathit{vars}(B_1)$, is a subconjunction of Catas_k, and
  - (ii) for every catamorphism atom $\mathit{cata}_j(X,T_j,Y_j)$ in $B_2\vartheta$, there is no catamorphism atom in Catas_k of the form $\mathit{cata}_j(V,T_j,W)$ (i.e., there is no catamorphism atom with the same predicate acting on the same ADT variable T_j);
- add the conjunction $B_2\vartheta$ to the body of C.

Then, by the *catamorphism addition rule*, we get the new set $S_{i+1}=(S_i\setminus\{C\})\cup\{E\}$.

Example 5.
In our *double* example, by applying the catamorphism addition rule to clause $E1$, we add the catamorphism $\mathit{listcount}(A,H,I)$, and we get:

$E2.\ \mathit{new}1(A,B,C,E,F)\leftarrow \mathit{listcount}(A,B,C),\ \mathit{listcount}(A,E,F),\ \mathit{listcount}(A,H,I),$
$\qquad \mathit{eq}(B,H),\ \mathit{append}(B,H,E)$

Thus, we get the new set of clauses $S_3=S_0\cup\{E2\}$. □

The following *folding rule* allows us to replace conjunctions of catamorphism atoms and program atoms by new program atoms whose predicates has been introduced in previous applications of the definition rule.

(R4) *Folding Rule.* Let C: $H\leftarrow c,\ \mathit{Catas}^C, A_1,\ldots,A_m$ be a clause in S_i, where either H is *false* or C has been obtained by the unfolding rule, possibly followed by the application of the catamorphism addition rule. Catas^C is a conjunction of catamorphisms and $A_1,\ldots,A_m$ are program atoms. For $k=1,\ldots,m$,

- let Catas^C_k be the conjunction of every catamorphism atom F in Catas^C such that $\mathit{adt\text{-}vars}(A_k)\cap \mathit{adt\text{-}vars}(F)\neq\emptyset$;
- let D_k: $H_k\leftarrow \mathit{Catas}^D_k, A_k$ be a clause in Defs_i (modulo variable renaming) such that Catas^C_k is a subconjunction of Catas^D_k.

Then, by *folding C using $D_1,\ldots,D_m$*, we derive clause E: $H\leftarrow c, H_1,\ldots,H_m$, and we get the new set of clauses $S_{i+1}=(S_i\setminus\{C\})\cup\{E\}$.

Example 6.
In order to fold clause $E2$ (see Example 5) according to the folding rule R4, we introduce for the program atoms $\mathit{append}(B,H,E)$ and $\mathit{eq}(B,H)$ that occur in the body of $E2$, the new definitions $D2$ and $D3$, respectively. Those new definitions are shown in Figure 2. Then, by folding clause $E2$ using $D2$ and $D3$, we get:

$E3.\ \mathit{new}1(A,B,C,E,F)\leftarrow \mathit{new}2(A,M,K,E,F,B,C),\ \mathit{new}3(A,M,K,B,C).$

https://doi.org/10.1017/S147106842400019X Published online by Cambridge University Press

Also, query 7 (see Figure 1) can be folded using definition $D1$, and we get:

$E4.$ *false* $\leftarrow C{=}2\,D{+}1,\ \ new1(A{,}E{,}F{,}G{,}C)$

Thus, $S_4 = (S_0 \setminus \{7\}) \cup \{E3, E4, D2, D3\}$. Then, we will continue by transforming the newly introduced definitions $D2$ and $D3$. □

The following Rule R5 is a new transformation rule that allows us: (i) to introduce new predicates by erasing ADT arguments from existing predicates, and (ii) to add atoms with these new predicates to the body of a clause.

(R5) *Erasure Addition Rule.* Let A be the atom $p(t_1, \ldots, t_k, u_1, \ldots, u_m)$, where $t_1, \ldots, t_k$ have (possibly distinct) basic sorts and $u_1, \ldots, u_m$ have (possibly distinct) ADT sorts. We define the *ADT-erasure* of A, denoted $\chi_{wo}(A)$, to be the atom $p\_woADTs(t_1, \ldots, t_k)$, where p_woADTs is a new predicate symbol. Let C: $H \leftarrow c, A_1, \ldots, A_n$ be a clause in S_i. Then, by the *erasure addition rule*, from C we derive the two new clauses:

$$\begin{array}{lll} \chi_{wo}(H)\leftarrow c, \chi_{wo}(A_1), \ldots, \chi_{wo}(A_n), & \text{denoted } \chi_{wo}(C), & \text{and} \\ H\leftarrow c, A_1, \chi_{wo}(A_1), \ldots, A_n, \chi_{wo}(A_n), & \text{denoted } \chi_{w\&wo}(C), & \end{array}$$

and we get the new set of clauses

$$S_{i+1} = \{\chi_{w\&wo}(C) \mid C \in S_i\} \ \cup\ \{\chi_{wo}(C) \mid C \text{ is a clause in } S_i \text{ whose head is not } \mathit{false}\}.$$

Example 7.
Let us consider clause $E3$ of Example 6. We have that:

$$\begin{array}{ll} \chi_{wo}(new1(A, B, C, E, F)) & = new1\_woADTs(A, C, F), \\ \chi_{wo}(new2(A, M, K, E, F, B, C)) & = new2\_woADTs(A, K, F, C), \\ \chi_{wo}(new3(A, M, K, B, C)) & = new3\_woADTs(A, K, C). \end{array}$$

Thus, from clause $E3$, by erasure addition we get clauses 12 and 18 of Figure 2. □

The following theorem is a consequence of well-known results for CHC transformations (see, for instance, the papers cited in a recent survey (De Angelis et al., 2022)).

Theorem 1 (Correctness of the Rules).
Let $S_0 \Rightarrow S_1 \Rightarrow \ldots \Rightarrow S_n$ be a transformation sequence using Rules R1–R5. Then, S_0 is satisfiable if and only if S_n is satisfiable.

Proof.
The proof consists in showing that Rules R1–R5 presented earlier in this section can be derived from the transformation rules considered in previous work (De Angelis et al., 2022) and proved correct based on results by Tamaki and Sato (Tamaki and Sato, 1986) for logic programs and Etalle and Gabbrielli (Etalle and Gabbrielli, 1996) for constraint logic programs. Below we will recall these transformation rules.

Let us first introduce the notion of *stratification* for a set of clauses (Lloyd, 1987). Let $\mathbb{N}$ be the set of the natural numbers and *Pred* be the set of the predicate names. A *level mapping* is a function λ: *Pred*$\rightarrow\mathbb{N}$. For every predicate p, the natural number $\lambda(p)$ is said to be the *level* of p. Level mappings are extended to atoms by stating that the level $\lambda(A)$ of an atom A is the level of its predicate symbol. A clause $H \leftarrow c, A_1, \ldots, A_n$ is *stratified* *with respect to* λ if either H is *false* or, for $i{=}1,\ldots,n$, $\lambda(H)\geq\lambda(A_i)$. A set P of CHCs is *stratified with respect to* λ if all clauses of P are stratified with respect to λ.

A *DUFR-transformation sequence from* S_0 *to* S_n is a sequence $S_0 \Rrightarrow S_1 \Rrightarrow \ldots \Rrightarrow S_n$ of sets of CHCs such that, for $i{=}0,\ldots,n{-}1$, S_{i+1} is derived from S_i, denoted $S_i \Rrightarrow S_{i+1}$, by applying one of the following rules: (i) Rule D, (ii) Rule U, (iii) Rule F, and (iv) Rule G. (To avoid confusion with Rules R1–R5 presented earlier in this section, in this proof we use the letters D, U, F, and G to identify the rules presented in previous work (De Angelis et al., 2022).) We assume that the initial set S_0 is stratified with respect to a given level mapping λ.

(Rule D) Let D be the clause $\mathit{newp}(X_1,\ldots,X_k) \leftarrow c, A_1,\ldots,A_m$, where: (1) *newp* is a predicate symbol not occurring in the sequence $S_0 \Rrightarrow S_1 \Rrightarrow \ldots \Rrightarrow S_i$ constructed so far, (2) c is a constraint, (3) the predicate symbols of $A_1,\ldots,A_m$ occur in S_0, and (4) $\{X_1,\ldots,X_k\} \subseteq \mathit{vars}(\{c, A_1,\ldots,A_m\})$. Then, by Rule D, we get $S_{i+1} = S_i \cup \{D\}$. We define the level mapping λ of *newp* to be equal to $\mathit{max}\,\{\lambda(A_i) \mid i=1,\ldots,m\}$.

For any $i\geq 0$, we denote by Defs_i the set of clauses introduced by Rule D during the construction of $S_0 \Rrightarrow S_1 \Rrightarrow \ldots \Rrightarrow S_i$.

Rule U consists in an application of the one-step unfolding of Definition 2.

(Rule U) Let C: $H \leftarrow c, G_L, A, G_R$ be a clause in S_i, where A is an atom. Then, by applying Rule U to C with respect to A, we get $S_{i+1} = (S_i \setminus \{C\}) \cup \mathit{Unf}(C, A, S_0)$.

(Rule F) Let C: $H \leftarrow c, G_L, Q, G_R$ be a clause in S_i, and let D: $K \leftarrow d, B$ be a variant of a clause in Defs_i. Suppose that: (1) either H is *false* or $\lambda(H)\geq\lambda(K)$, and (2) there exists a substitution ϑ such that $Q{=}B\vartheta$ and $\mathbb{D} \models \forall(c \rightarrow d\vartheta)$. Then, by applying Rule F to C using D, we derive clause E: $H \leftarrow c, G_L, K\vartheta, G_R$, and we get $S_{i+1} = (S_i \setminus \{C\}) \cup \{E\}$.

In the next Rule R, called *goal replacement*, and in the rest of the proof, by $\mathit{Definite}(S_0)$ we denote the set of definite clauses belonging to S_0.

(Rule R) Let C: $H \leftarrow c, c_1, G_L, G_1, G_R$ be a clause in S_i. Suppose that the following two conditions hold:

(R.1) $M(\mathit{Definite}(S_0) \cup \mathit{Defs}_i) \models \forall\,((\exists T_1.\, c_1 {\wedge} G_1) \leftrightarrow (\exists T_2.\, c_2 {\wedge} G_2))$, and

(R.2) either H is *false* or, for every atom A occurring in G_2 and not in G_1, $\lambda(H){>}\lambda(A)$ where:

$T_1 = \mathit{vars}(\{c_1, G_1\}) \setminus \mathit{vars}(\{H, c, G_L, G_R\})$, and

$T_2 = \mathit{vars}(\{c_2, G_2\}) \setminus \mathit{vars}(\{H, c, G_L, G_R\})$.

Then, by Rule R, in clause C we *replace* c_1, G_1 *by* c_2, G_2, and we derive clause D: $H \leftarrow c, c_2, G_L, G_2, G_R$. We get $S_{i+1} = (S_i \setminus \{C\}) \cup \{D\}$.

The following result guarantees that, for any DUFR-transformation sequence $S_0 \Rrightarrow S_1 \Rrightarrow \ldots \Rrightarrow S_n$ satisfying Condition (C), S_0 and S_n are equisatisfiable (Tamaki and Sato, 1986; Etalle and Gabbrielli, 1996; De Angelis et al., 2022).

Theorem 2 (Correctness of the DUFR-Transformation Rules).
Let $S_0 \Rrightarrow S_1 \Rrightarrow \ldots \Rrightarrow S_n$ be a DUFR-transformation sequence. Suppose that the following condition holds:

https://doi.org/10.1017/S147106842400019X Published online by Cambridge University Press

(C) for i=1, ..., n−1, if $S_i \Rrightarrow S_{i+1}$ by folding a clause in S_i using a definition D: $H \leftarrow c, B$ in $Defs_i$, then, for some $j \in \{1, \ldots, i-1, i+1, \ldots, n-1\}$, $S_j \Rrightarrow S_{j+1}$ by unfolding D with respect to an atom A such that $\lambda(H) = \lambda(A)$.

Then,

(1) for $i = 1, \ldots, n$, $M(\mathit{Definite}(S_0) \cup \mathit{Defs}_i) = M(\mathit{Definite}(S_i))$, and
(2) S_0 is satisfiable if and only if S_n is satisfiable.

Now, we will show that each application of Rules R1–R5 can be obtained by one or more applications of Rules D, U, F, R. Furthermore, for any transformation sequence $S_0 \Rrightarrow S_1 \Rrightarrow \ldots \Rrightarrow S_n$ constructed using Rules R1–R5, there exists a DUFR-transformation sequence $S_0 \Rrightarrow T_0 \Rrightarrow \ldots \Rrightarrow T_r \Rrightarrow S_n$ satisfying Condition (C) of Theorem 2.

In order to recast Rules R1–R5 in terms of Rules D, U, F, and R, we first introduce a suitable level mapping λ defined as follows: for any predicate q, (i) $\lambda(q)$=2, if q is a program predicate of the initial set of clauses or a new program predicate introduced by Rule R1, and (ii) $\lambda(q)$=1, if q is a catamorphism predicate, and (iii) $\lambda(q)$=0, if q is a new predicate symbol introduced by Rule R5. We have that the initial set S_0 of CHCs is stratified with respect to λ. Let us first consider the four Rules R1–R4.

- Rule R1 is a particular case of Rule D, where in the body of clause D, (i) the constraint c is absent, (ii) exactly one atom among $A_1, \ldots, A_m$ is a program atom, (iii) all other atoms are catamorphism atoms, and (iv) $\{X_1, \ldots, X_k\} = \mathit{vars}(\{A_1, \ldots, A_m\})$. By our definition of the level mapping, $\lambda(\mathit{newp}) = 2$, as one of the A_i's is a program atom.
- Rule R2 consists of applications of Rules U and R. Indeed, in R2, (i) Steps 1 and 2 are applications of Rule U where P is S_0, and (ii) Step 3 is an application of Rule R. To see Point (ii), note that every catamorphism *cata* is, by definition, a functional predicate (see Section 2), and hence $M(\mathit{Definite}(S_0)) \models \forall(\mathit{cata}(X,T,Y1) \wedge \mathit{cata}(X,T,Y2) \rightarrow Y1{=}Y2)$. Thus, for any $i \geq 0$,

  $$M(\mathit{Definite}(S_0) \cup \mathit{Defs}_i) \models \forall(\mathit{cata}(X,T,Y1) \wedge \mathit{cata}(X,T,Y2) \leftrightarrow \\ Y1{=}Y2 \wedge \mathit{cata}(X,T,Y1))$$

  that is, Condition (R.1) of Rule R holds. Also Condition (R.2) holds, as the head H of the clause has a predicate *newp* introduced by definition, and hence $\lambda(\mathit{newp}) = 2$, while we have stipulated that $\lambda(\mathit{cata}) = 1$.
- Rule R3 consists of applications of Rule R. Indeed, R3 adds to the body of a clause C (zero or more) catamorphism atoms $\mathit{cata}_j(X, T_j, Y_j)$ such that no variable in the tuple Y_j occurs in C. The assumption that catamorphisms are total functions enforces that $M(\mathit{Definite}(S_0)) \models \forall X, T_j\ \exists Y_j.\ \mathit{cata}_j(X, T_j, Y_j)$, and hence

  $$M(\mathit{Definite}(S_0) \cup \mathit{Defs}_i) \models \forall(\mathit{true} \leftrightarrow \exists Y_j.\ \mathit{cata}_j(X, T_j, Y_j))$$

  that is, Condition (R.1) of Rule R holds. Also Condition (R.2) holds, as the head H of clause C is either *false* or a program atom. In the latter case $\lambda(H) = 2$, while we have stipulated that $\lambda(\mathit{cata}_j) = 1$.
- Rule R4 consists of applications of Rules R and F. Indeed, an application of Rule R3 is equivalent to the following for-loop of applications of Rules R and F: for

k=1, ..., m, first, (i) the addition of the catamorphism atoms occurring (modulo variable renaming) in $Catas_k^D$ and not in its subconjunction $Catas_k^C$ (as mentioned above, this catamorphism addition is an instance of Rule R), and then, (ii) the application of Rule F, thereby replacing the conjunction $(Catas_k^D, A_k)$ by H_k.

Therefore, for any transformation sequence $S_0 \Rrightarrow S_1 \Rrightarrow \ldots \Rrightarrow S_i$ constructed using Rules R1–R4, there exists a DUFR-transformation sequence $S_0 \Rrightarrow T_0 \Rrightarrow \ldots \Rrightarrow T_r \Rrightarrow S_i$. When applying Rule R4 to a clause C during the construction of $S_0 \Rrightarrow S_1 \Rrightarrow \ldots \Rrightarrow S_i$, either the head of C is *false* or C has been obtained by the unfolding rule (possibly followed by catamorphism addition). This implies that in $S_0 \Rrightarrow T_0 \Rrightarrow \ldots \Rrightarrow T_r \Rrightarrow S_i$ we have that Condition (C) of Theorem 2 holds. Thus, by Theorem 2 we get: $M(\mathit{Definite}(S_0) \cup \mathit{Defs}_i) = M(\mathit{Definite}(S_i))$.

Now, suppose that we apply Rule R5 to the set S_i of clauses. We have that, for every predicate p occurring in S_i,

$$M(\mathit{Definite}(S_i) \cup \chi_{wo}(S_i)) \models \forall(p(X_1, \ldots, X_k, Y_1, \ldots, Y_m) \rightarrow p\_woADTs(X_1, \ldots, X_k)) \;(\dagger)$$

where $\chi_{wo}(S_i) = \{\chi_{wo}(C) \mid C$ is a clause in S_i whose head is not *false*$\}$. Now, it is the case that an application of Rule R5 is realized by a sequence of applications of Rule R. Indeed, for each addition of an atom $p\_woADTs(t_1, \ldots, t_k)$ to the body of a clause C by R5, Condition (R.1) holds, as the above relation (†) is equivalent to:

$$M(\mathit{Definite}(S_i) \cup \chi_{wo}(S_i)) \models$$
$$\forall(p(X_1, \ldots, X_k, Y_1, \ldots, Y_m) \leftrightarrow (p(X_1, \ldots, X_k, Y_1, \ldots, Y_m) \wedge p\_woADTs(X_1, \ldots, X_k)))$$

and $M(\mathit{Definite}(S_i) \cup \chi_{wo}(S_i)) = M(\mathit{Definite}(S_0 \cup \chi_{wo}(S_i)) \cup \mathit{Defs}_i)$, because the predicates in $\chi_{wo}(S_i)$ do not occur in $S_0, \ldots, S_i$. Also Condition (R.2) holds, because the head H of C is either *false* or $\lambda(H) \geq 1$ and $\lambda(p\_woADTs)$=0.

Therefore, for any transformation sequence $S_0 \Rrightarrow S_1 \Rrightarrow \ldots \Rrightarrow S_n$ constructed using Rules R1–R5, there exists a DUFR-transformation sequence $S_0 \Rrightarrow T_0 \Rrightarrow \ldots \Rrightarrow T_r \Rrightarrow S_n$. Then, by Theorem 2, we get that S_0 is satisfiable if and only if S_n is satisfiable. □

4.3 The transformation algorithm $\mathcal{T}_{abs}$

The set of the new predicate definitions needed during the execution of the transformation algorithm $\mathcal{T}_{abs}$ is not given in advance. In general, that set depends on: (i) the initial set P of CHC clauses, (ii) the given query Q specifying the property of interest to be proved, and (iii) the given catamorphic abstraction specification α for P. As we will see, we may compute that set of new definitions as the least fixpoint of an operator, called $\tau_{P\cup\{Q\},\alpha}$, which transforms a given set Δ of predicate definitions into a new set Δ' of predicate definitions. First, we need the following notions.

Two definitions D_1 and D_2 are said to be *equivalent*, denoted $D_1 \equiv D_2$, if they can be made identical by performing the following transformations: (i) renaming of the head predicate, (ii) renaming of the variables, (iii) reordering of the variables in the head, and (iv) reordering of the atoms in the body. We leave it to the reader to check that the results presented in this section are indeed independent of the choice of a specific definition in its equivalence class.

A set Δ of definitions is said to be *monovariant* if, for each program predicate p, in Δ there is at most one definition having an occurrence of p in its body. The transformation algorithm $\mathcal{T}_{abs}$ and the operator $\tau_{P\cup\{Q\},\alpha}$ work on monovariant sets of definitions and are defined by means of the *Define*, *Unfold*, *AddCata*, *Fold*, and *AddErasure* functions defined in Figure 3.

In the definition of the *Define* function we assume that, for each clause C in *Cls* and each catamorphism atom *Cata* in the body of C, there is a program atom A in the body of C such that $adt\text{-}vars(Cata) \subseteq adt\text{-}vars(A)$. If A is absent for a catamorphism atom having the ADT variable X of sort τ, in order to comply with our assumption, we add to the body of C a program atom $true_\tau(X)$ that is defined on the (possibly recursive) structure of sort τ and holds for every X of sort τ. For instance, for the sort $list(int)$, the program atom $true_{list(int)}(X)$ will be defined by the two clauses $true_{list(int)}([\,])$ and $true_{list(int)}([H|T]) \leftarrow true_{list(int)}(T)$, where H is an integer variable. Note that, by adding to clause C the atom $true_\tau(X)$, we get a clause equivalent to C.

Definition 3 (Domain of Definitions).
We denote by $\mathcal{D}$ a maximal set of definitions such that

(D1) for every definition $newp(X_1,\ldots,X_k) \leftarrow Catas, A$ in $\mathcal{D}$, for every ADT variable X_i occurring in the program atom A, for each catamorphism predicate cata in the conjunction Catas of catamorphism atoms, at most one catamorphism atom of the form $cata(\ldots,X_i,\ldots)$ occurs in Catas, and

(D2) $\mathcal{D}$ does not contain equivalent definitions.
It follows directly from our assumptions that $\mathcal{D}$ is a finite set.

Now we define a partial order ($\sqsubseteq$), a join operation ($\sqcup$) and a meet operation ($\sqcap$) for definitions and also for monovariant subsets of definitions in $\mathcal{D}$.

Definition 4.
Let D_1: $newp1(U_1) \leftarrow Catas_1, Catas, A$ and D_2: $newp2(U_2) \leftarrow Catas_2, Catas, A$ be two definitions in $\mathcal{D}$ for the same program atom A, where $Catas$, $Catas_1$, and $Catas_2$ are conjunctions of catamorphism atoms. We assume that the variables in D_1 and D_2 have been renamed and the atoms in their bodies have been reordered so that $(Catas, A)$ is the maximal common subconjunction of atoms in their bodies, that is, there exists no atom *Cata* in $Catas_1$ and no variant of D_2 of the form $newp2(U_2') \leftarrow Catas_2', Catas, A$, such that *Cata* is an atom in $Catas_2'$.

(i) D_2 is an *extension* of D_1, written $D_1 \sqsubseteq D_2$, if $Catas_1$ is the empty conjunction;

(ii) By $D_1 \sqcup D_2$ we denote the definition D_3: $newp3(U_3) \leftarrow Catas_1, Catas_2, Catas, A$, where U_3 is a tuple consisting of the distinct variables occurring in (U_1,U_2);

(iii) By $D_1 \sqcap D_2$ we denote the definition D_3: $newp3(U_3) \leftarrow Catas, A$, where U_3 is a tuple consisting of the variables occurring in both U_1 and U_2.

Let Δ_1 and Δ_2 be two monovariant subsets of $\mathcal{D}$.

(iv) Δ_2 is an *extension* of Δ_1, written $\Delta_1 \sqsubseteq \Delta_2$, if for each D_1 in Δ_1 there exists D_2 in Δ_2 such that $D_1 \sqsubseteq D_2$;

(v) $\Delta_1 \sqcup \Delta_2 = \{D \mid D$ is the only definition in $\Delta_1 \cup \Delta_2$ for some program atom in $\Delta_1 \cup \Delta_2\} \,\cup$
$\{D_1 \sqcup D_2 \mid D_1$ and D_2 are definitions for the same program atom in Δ_1 and Δ_2, respectively$\}$;

https://doi.org/10.1017/S147106842400019X Published online by Cambridge University Press

Given a set *Cls* of clauses and a monovariant set Δ of definitions,
$\mathit{Define}(\mathit{Cls},\Delta)$ returns a monovariant set Δ' of new definitions computed as follows.

$\Delta' := \Delta$;
for each clause $H \leftarrow c, G$ in *Cls*, where goal G contains at least one ADT variable **do**
  for each program atom A in G **do**
    $\mathit{Catas}_A := \{F \mid F$ is a catamorphism atom in G and $\mathit{adt\text{-}vars}(F) \subseteq \mathit{adt\text{-}vars}(A)\}$;
    if there is a clause $D\colon \mathit{newp}(U) \leftarrow B, A$ in Δ', for some conjunction B of catamorphism atoms
    then if Catas_A is *not* a subconjunction of B **then** // (**Extend**)
      by applying the definition rule R1, introduce the definition
      ExtD: $\mathit{extp}(V) \leftarrow B', A$, where: (i) *extp* is a new predicate symbol, (ii) B' is the conjunction of the distinct catamorphism atoms occurring either in B or in Catas_A, and (iii) $V = \mathit{vars}(\{B', A\})$;
      $\Delta' := (\Delta' \setminus \{D\}) \cup \{\mathit{ExtD}\}$;
    else by applying the definition rule R1, introduce the definition // (**Add**)
      NewD: $\mathit{newp}(V) \leftarrow \mathit{Catas}_A, A$, where: (i) *newp* is a new predicate symbol, and (ii) $V = \mathit{vars}(\{\mathit{Catas}_A, A\})$;
      $\Delta' := \Delta' \cup \{\mathit{NewD}\}$;

Given a set $\Delta = \{D_i \mid 0 \le i \le n\}$ of definitions and a set P of definite clauses,
$\mathit{Unfold}(\Delta, P) = \bigcup_{i=1}^{n} \mathit{UnfD}_i$, where UnfD_i is the set of clauses derived by applying the unfolding rule R2 to clause D_i.

Given a set $\mathit{Cls} = \{C_i \mid 0 \le i \le n\}$ of clauses and a catamorphic abstraction specification α,
$\mathit{AddCata}(\mathit{Cls}, \alpha) = \{E_i \mid 0 \le i \le n$ and E_i is obtained from C_i by applying the catamorphism addition rule R3 using $\alpha\}$.

Given a set $\mathit{Cls} = \{C_i \mid 0 \le i \le n\}$ of clauses and a monovariant set Δ of definitions,
$\mathit{Fold}(\mathit{Cls}, \Delta) = \{E_i \mid 0 \le i \le n$ and E_i is obtained from C_i by applying the folding rule R4 using definitions in $\Delta\}$.

Given a set *Cls* of clauses by applying the erasure addition rule R5,
$\mathit{AddErasure}(\mathit{Cls}) = \{\chi_{wo}(C) \mid C$ is a clause in *Cls* whose head is not $\mathit{false}\} \cup \{\chi_{w\&wo}(C) \mid C \in \mathit{Cls}\}$.

Fig. 3. The *Define*, *Unfold*, *AddCata*, *Fold*, and *AddErasure* functions.

(vi) $\Delta_1 \sqcap \Delta_2 = \{D_1 \sqcap D_2 \mid D_1$ and D_2 are the definitions for the same program atom in Δ_1 and Δ_2, respectively$\}$.

Let $\mathcal{P}_m(\mathcal{D})$ denote the set of monovariant subsets of $\mathcal{D}$. We have that $(\mathcal{P}_m(\mathcal{D}), \sqsubseteq, \sqcup, \sqcap)$ is a lattice and, since $\mathcal{D}$ is a finite set, it is also a complete lattice. We define the operator $\tau_{P\cup\{Q\},\alpha}\colon \mathcal{P}_m(\mathcal{D}) \to \mathcal{P}_m(\mathcal{D})$ as follows:

$$\tau_{P\cup\{Q\},\alpha}(\Delta) =_{def} \mathit{Define}(\mathit{AddCata}(\mathit{Unfold}(\Delta, P) \cup \{Q\}, \alpha), \Delta)$$

Now, we show that the operator $\tau_{P\cup\{Q\},\alpha}$ is a well defined function from $\mathcal{P}_m(\mathcal{D})$ to itself, that is, for any $\Delta \in \mathcal{P}_m(\mathcal{D})$, the set $\Delta' = \tau_{P\cup\{Q\},\alpha}(\Delta)$ is an element of $\mathcal{P}_m(\mathcal{D})$.

First, note that: (i) the *Define* function introduces (see the (Add) case) a new definition for a program predicate only if no definition for that predicate already belongs to Δ, and (ii) *Define* replaces (see the (Extend) case) a definition for a program predicate by a new

https://doi.org/10.1017/S147106842400019X Published online by Cambridge University Press

definition for the same predicate. Thus, if Δ is monovariant, so is Δ'. Moreover, no two equivalent clauses will belong to Δ' (see Point (D2) of Definition 3).

Note also that, due to the definition of function *AddCata* (see, in particular, Point (ii) of Rule R3 applied by that function), Point (D1) of Definition 3 holds, and in particular, for every ADT variable X_i in the body of any new definition in Δ', and for every catamorphism predicate *cata*, there is at most one catamorphism atom of the form $cata(\ldots, X_i, \ldots)$.

Lemma 1 (Existence and Uniqueness of the Fixpoint of $\tau_{P\cup\{Q\},\alpha}$*).* The operator $\tau_{P\cup\{Q\},\alpha}$ is monotonic on the finite lattice $\mathcal{P}_m(\mathcal{D})$. Thus, it has a least fixpoint $\mathit{lfp}(\tau_{P\cup\{Q\},\alpha})$, also denoted τ_{fix}, which is equal to $\tau^n_{P\cup\{Q\},\alpha}(\emptyset)$, for some natural number n.

Proof.
In order to prove the monotonicity of $\tau_{P\cup\{Q\},\alpha}$, let us assume that Δ_1 and Δ_2 are two sets of monovariant definitions in $\mathcal{P}_m(\mathcal{D})$, with $\Delta_1 \sqsubseteq \Delta_2$. Let $D_1 \in \tau_{P\cup\{Q\},\alpha}(\Delta_1)$ be a definition for program atom A. We consider two cases.

(*Case* 1) There is no definition for A in Δ_1. Then, by construction, according to the *Define* function (see Figure 3), D_1 can be viewed as the result of a sequence of join operations of the form: $E_0 \sqcup E_1 \sqcup \ldots \sqcup E_n$, with $n{\geq}0$, where: (1) clause E_0 has been obtained by the (Add) case of *Define*, and (2) for $i{=}1,\ldots,n$, clause $E_0 \sqcup \ldots \sqcup E_i$ is a clause obtained by the (Extend) case of *Define* from clause $E_0 \sqcup \ldots \sqcup E_{i-1}$. In particular, for all $i{=}0,\ldots,n$, clause E_i is a clause of the form $\mathit{newp}_i(V_i) \leftarrow \mathit{Catas}_i, A$ obtained from a clause $H \leftarrow c, G$ (here and below in this proof H may be *false*) in $\mathit{AddCata}(\mathit{Unfold}(\Delta_1, P) \cup \{Q\}, \alpha)$ such that A is a program atom in G and Catas_i is the conjunction of all catamorphism atoms F in G with $\mathit{adt\text{-}vars}(F) \subseteq \mathit{adt\text{-}vars}(A)$.

(*Case* 2) There is a definition E_0 for A in Δ_1. Then, similarly to Case 1, by construction, $D_1 = E_0 \sqcup \ldots \sqcup E_n$, where, for $i{=}1,\ldots,n$, with $n{\geq}0$, $E_0 \sqcup \ldots \sqcup E_i$ is a clause obtained by the (Extend) case of *Define*.

Now, since $\Delta_1 \sqsubseteq \Delta_2$, for each clause $H \leftarrow c, G$ in $\mathit{AddCata}(\mathit{Unfold}(\Delta_1, P) \cup \{Q\}, \alpha)$, there exists a clause $H \leftarrow c, C, G$ in the set of clauses $\mathit{AddCata}(\mathit{Unfold}(\Delta_2, P) \cup \{Q\}, \alpha)$, where C is a conjunction of catamorphism atoms, and then, by construction, $\mathit{Define}(\mathit{AddCata}(\mathit{Unfold}(\Delta_2, P) \cup \{Q\}, \alpha), \Delta_2)$ contains, for $i{=}1,\ldots,n$, a clause E'_i, with $E_i \sqsubseteq E'_i$. Then, there exists $D_2 \in \tau_{P\cup\{Q\},\alpha}(\Delta_2)$ such that $D_1{=}(E_0 \sqcup \ldots \sqcup E_n) \sqsubseteq (E'_0 \sqcup \ldots \sqcup E'_n) \sqsubseteq (E'_0 \sqcup \ldots \sqcup E'_n \sqcup F_1 \sqcup \ldots \sqcup F_r) = D_2$, with $r \geq 0$. (Note that, since $\Delta_1 \sqsubseteq \Delta_2$, in $\mathit{AddCata}(\mathit{Unfold}(\Delta_2, P) \cup \{Q\}, \alpha)$ there may be clauses that are derived from definitions in Δ_2 that are not extensions of definitions in Δ_1. In the bodies of those clauses there may be some variants of A that determine r extra applications of the (Extend) case of *Define*.) Therefore, by Definition 4, $\tau_{P\cup\{Q\},\alpha}(\Delta_1) \sqsubseteq \tau_{P\cup\{Q\},\alpha}(\Delta_2)$.

Thus, $\tau_{P\cup\{Q\},\alpha}$ is monotonic with respect to $\sqsubseteq$. Since $\mathcal{P}_m(\mathcal{D})$ is a finite, hence complete, lattice, $\tau_{P\cup\{Q\},\alpha}$ has a least fixpoint $\mathit{lfp}(\tau_{P\cup\{Q\},\alpha})$, which can be computed as $\tau^n_{P\cup\{Q\},\alpha}(\emptyset)$, for some natural number n. □

Now, we define our transformation algorithm $\mathcal{T}_{abs}$ as follows:

$$\mathcal{T}_{abs}(P \cup \{Q\}, \alpha) = \mathit{AddErasure}(\mathit{Fold}(\mathit{AddCata}(\mathit{Unfold}(\tau_{\mathit{fix}}, P) \cup \{Q\},\ \alpha), \tau_{\mathit{fix}}))$$

https://doi.org/10.1017/S147106842400019X Published online by Cambridge University Press

The termination of $\mathcal{T}_{abs}$ follows immediately from the fact that the functions *Unfold*, *AddCata*, *Fold*, and *AddErasure* terminate and the least fixpoint τ_{fix} is computed in a finite number of steps (see Lemma 1). Thus, by the correctness of the transformation rules (see Theorem 1), we get the following result.

Theorem 3 (Total Correctness of Algorithm $\mathcal{T}_{abs}$). $\mathcal{T}_{abs}$ terminates for any set P of definite clauses, query Q, and catamorphic abstraction specification α. Also, $P \cup \{Q\}$ is satisfiable if and only if $\mathcal{T}_{abs}(P \cup \{Q\}, \alpha)$ is satisfiable.

Finally, we would like to comment on the fact that our transformation algorithm $\mathcal{T}_{abs}$ introduces a monovariant set of definitions. Other definition introduction policies could have been considered. In particular, one could introduce more than one definition for each program predicate, thus producing a *polyvariant* set of definitions. The choice between monovariant and polyvariant sets of definitions has been subject to ample discussion in the literature (De Angelis et al., 2022) and both have advantages and disadvantages. We will show in the next section that our technique performs quite well in our benchmark. However, we leave a more accurate experimental evaluation to future work.

5 Implementation and experimental evaluation

In this section we provide some details on the implementation of algorithm $\mathcal{T}_{abs}$, and on its experimental evaluation.

Implementation. We have implemented algorithm $\mathcal{T}_{abs}$ in a tool, called VeriCaT$_{abs}$, based on VeriMAP (De Angelis et al., 2014), which is a system for transforming CHCs. In order to check satisfiability of sets of CHCs (before and after their transformation) we have used the following two solvers: (i) Eldarica (v. 2.0.9) (Hojjat and Rümmer, 2018), and (ii) Z3 (v. 4.12.2) (de Moura and Bjørner, 2008) with the SPACER engine (Komuravelli et al., 2016) and the *global guidance* option (Krishnan et al., 2020).

The tool VeriCaT$_{abs}$ manipulates clauses as indicated in the following three phases.

(*Phase* 1) A pre-processing phase. In this phase VeriCaT$_{abs}$ produces a catamorphic abstraction specification α starting from: (i) a given set P of CHCs, and (ii) the catamorphic abstractions for the ADTs occurring in P. For instance, in the case of our introductory example *double* (see Figure 1), Phase 1 produces the catamorphic abstraction specifications for *double*, *eq*, and *append* we have listed in Example 1, starting from clauses 1–6 and the catamorphic abstraction $cata_{list(int)} =_{def} listcount(X, L, N)$,

In the following example, referring to a treesort algorithm, we present the VeriCaT$_{abs}$ syntax for representing: (i) the catamorphic abstractions given in input, using the directive `cata_abs`, and (ii) the catamorphic abstraction specifications produced in output, after Phase 1, using the directive `spec`.

Example 8.
Let `treesort(L,S)` and `visit(T,L)` be two atoms included in a CHC encoding of the treesort algorithm. The atom `treesort(L,S)` holds if and only if `S` is the list of integers obtained by applying the treesort algorithm to the list `L` of integers. The auxiliary atom

https://doi.org/10.1017/S147106842400019X Published online by Cambridge University Press

`visit(T,L)` holds if and only if `L` is the list of integers obtained by a depth first visit of the tree `T` with integers at its nodes. The catamorphic abstractions for the ADT sorts *list*(*int*) and *tree*(*int*) used by our tool VeriCaT$_{abs}$ during Phase 1, are as follows:

```
:- cata_abs list(int) ==> listcount(X,L,C).
:- cata_abs tree(int) ==> treecount(X,T,C).
```

The catamorphisms `listcount(X,L,B)` and `treecount(X,T,A)` count the occurrences of the integer `X` in the list `L` and in the tree `T`, respectively. In general, the directive `cata_abs` for a sort τ is as follows:

`:- cata_abs` τ `==>` catamorphisms acting on τ.

For the program predicates `treesort` and `visit`, the catamorphic abstraction specifications produced by VeriCaT$_{abs}$ after Phase 1, are as follows:

```
:- spec treesort(L,S) ==> X=Y, listcount(X,S,A), listcount(Y,L,B).
:- spec visit(T,L) ==> X=Y, treecount(X,T,A), listcount(Y,L,B).
```

Note that both the tree catamorphism `treecount(X,T,A)` and the list catamorphism `listcount(Y,L,B)` occur in the catamorphic specification for `visit(T,L)`. □

(*Phase* 2) A fold/unfold transformation phase. In this phase VeriCaT$_{abs}$ computes the fixpoint τ_{fix} and the set T_w of clauses, which is $\mathit{Fold}(\mathit{AddCata}(\mathit{Unfold}(\tau_{\mathit{fix}}, P) \cup \{Q\},\ \alpha), \tau_{\mathit{fix}})$. For the *double* introductory example (see Figure 1), we have that P is the set $\{1, \ldots, 6\}$ of clauses, query Q is clause 7, and α is the set of catamorphic abstraction specifications produced at Phase 1 (see Example 1). Now, τ_{fix} is the set $\{D1, D2, D3, D4\}$ of definitions listed in Figure 2 and the set T_w is as follows:

$$
\begin{array}{l}
\mathit{false} \leftarrow C{=}2D+1,\ \ \mathit{new}1(A,E,F,G,C)\\
\mathit{new}1(A,B,C,E,F) \leftarrow \mathit{new}2(A,M,K,E,F,B,C),\ \ \mathit{new}3(A,M,K,B,C)\\
\mathit{new}2(A,B,C,B,C,[\,],G) \leftarrow G{=}0,\ \ \mathit{new}4(A,B,C)\\
\mathit{new}2(A,B,C,[E|F],G,[E|J],K) \leftarrow G{=}\mathit{ite}(A{=}E,N{+}1,N),\ \ K{=}\mathit{ite}(A{=}E,P{+}1,P),\\
\qquad\qquad\qquad \mathit{new}2(A,B,C,F,N,J,P)\\
\mathit{new}3(A,B,C,B,C) \leftarrow \mathit{new}4(A,B,C)\\
\mathit{new}4(A,[\,],B) \leftarrow B{=}0\\
\mathit{new}4(A,[B|C],D) \leftarrow D{=}\mathit{ite}(A{=}B,F{+}1,F),\ \ \mathit{new}4(A,C,F)
\end{array}
$$

(*Phase* 3) A post-processing phase. In this phase, VeriCaT$_{abs}$ produces the following two additional sets of clauses by applying the *AddErasure* function to T_w:

(i) $T_{wo} = \{\chi_{wo}(C) \mid C \text{ is a clause in } T_w\}$, that is, T_{wo} is made out of the clauses in T_w where every atom with ADT arguments has been replaced by its corresponding atom without ADT arguments, and

(ii) $T_{w\&wo} = \{\chi_{w\&wo}(C) \mid C \text{ is a clause in } T_w\} \cup \overline{T}_{wo}$, that is, $T_{w\&wo}$ is made out of the clauses in *either* (ii.1) T_w, where every atom *in the body* with ADT arguments is paired with its corresponding atom without ADT arguments, *or* (ii.2) $\overline{T}_{wo} = \{\chi_{wo}(C) \mid C \text{ is a clause in } T_w \text{ whose head is not } \mathit{false}\}$.

$T_{w\&wo}$ is, indeed, the set of clauses computed by our transformation algorithm $\mathcal{T}_{abs}$. The other two sets T_w and T_{wo}, produced by VeriCaT$_{abs}$, will be used for comparing and analyzing the features of $T_{w\&wo}$, as we do in the experimental evaluation below.

https://doi.org/10.1017/S147106842400019X Published online by Cambridge University Press

For our introductory example *double* (see Figure 1), at the end of Phase 3, VeriCaT$_{abs}$ produces the following two sets of clauses (clause numbers refer to Figure 2):

T_{wo}=$\{$*false* $\leftarrow C$=$2D$+1, *new*1_*woADTs*$(A,F,C)\} \cup \{18, \ldots, 23\}$, and
$T_{w\&wo} = \{11, \ldots, 23\}$.

The set $\{18, \ldots, 23\}$ of clauses is $\overline{T}_{wo}$ of Point (ii.2) above.

Experimental evaluation. Our benchmark consists of 228 sets of CHCs that encode properties of various sorting algorithms (such as bubblesort, heapsort, insertionsort, mergesort, quicksort, selectionsort, and treesort), and simple list and tree manipulation algorithms (such as appending and reversing lists, constructing permutations, deleting copies of elements, manipulating binary search trees). Properties of those algorithms are expressed via catamorphisms. Here is a non-exhaustive list of the catamorphisms we used: (i) *size*(L, S), (ii) *listmin*(L, Min), (iii) *listmax*(L, Max), and (iv) *sum*(L, Sum) computing, respectively, the size S of list L, the minimum Min, the maximum Max, and the sum Sum of the elements of list L, (v) *is_asorted*(L, BL), which holds with BL=*true* if and only if list L is ordered in weakly ascending order, (vi) *allpos*(L, B), which holds with B=*true* if and only if list L is made out of all positive elements, (vii) *member*(X, L, B), which holds with B=*true* if and only if X is an element of the list L, and (viii) *listcount*(X, L, N), which holds if and only if N is the number ($\geq$0) of occurrences of X in the list L. For some properties, we have used more than one catamorphism at a time and, in particular, for lists of integers, we have used the conjunction of *member* and *listcount*, and for different properties, we have also used the conjunction of *listmin*, *listmax*, and *is_asorted*, as already indicated in the paper.

A property holds if and only if its CHC encoding via a query Q is satisfiable, and a verification task consists in using a CHC solver to check the satisfiability of Q. When the given property holds for a set P of clauses, the solver should return *sat* and the property is said to be a *sat* property. Analogously, when a property does not hold, the solver should return *unsat* and the property is said to be an *unsat* property. In our benchmark, for each verification task of a *sat* property, we have considered a companion verification task whose CHCs have been modified so that the associated property is *unsat*. In particular, we have 114 *sat* properties and 114 *unsat* properties.

We have performed our experiments on an Intel(R) Xeon(R) Gold 6238R CPU 2.20 GHz with 221 GB RAM under CentOS with a timeout of 600s per verification task. The results of our experiments are reported in Table 1. The VeriCaT$_{abs}$ tool and the benchmarks are available at https://fmlab.unich.it/vericatabs.

Table 1 shows that, for each verification task, the transformation of the CHCs allows a very significant improvement of the performance of the Z3 solver and also an overall improvement of the Eldarica solver (notably for *sat* properties).

In particular, before CHC transformation, Z3 did not prove any of the 114 *sat* properties of our benchmark. After CHC transformation, Z3 proved 109 of them to be *sat* (see columns Z_1 and Z_3 of Table 1). The time cost of this improvement is very small. Indeed, most CHC transformations take well below 1.5s and only one of them takes a little more than 2s (for details, see column T, where each entry is the sum of the times taken for the individual transformation tasks of each row). The times taken by the solvers after

Table 1. *Properties proved by the solvers Eldarica and Z3* before *and* after *the transformation performed by algorithm $\mathcal{T}_{abs}$. In the* before *case, the input to the solver is the source set of clauses (src-columns), and in the* after *case, the input is $T_{w\&wo}$ ($T_{w\&wo}$-columns). The columns occur in pairs referring to the sat properties (s-columns) and the unsat properties (u-columns), respectively. The two T_w-columns and the two T_{wo}-columns refer to the input T_w and T_{wo}, respectively. The last column shows the time (in seconds) taken by $\mathcal{T}_{abs}$ as implemented by VeriCaT$_{abs}$.*

			Eldarica								Z3								Transf time
	Properties		*src*		$T_{w\&wo}$		T_w		T_{wo}		*src*		$T_{w\&wo}$		T_w		T_{wo}		
Programs	s	u	s	u	s	u	s	u	s	u	s	u	s	u	s	u	s	u	T
Append	4	4	0	3	3	3	3	4	3	4	0	4	4	4	4	4	4	4	6.4
Bubblesort	9	9	2	9	9	9	9	9	9	9	0	9	9	9	9	9	9	9	15.8
BinSearchTree	8	8	0	7	0	5	2	7	4	8	0	8	8	8	7	8	8	8	19.2
DeleteCopies	7	7	0	7	4	7	3	7	6	7	0	7	7	7	7	7	7	7	11.1
Heapsort	7	7	0	7	2	7	0	7	4	7	0	7	7	7	3	7	5	7	13.5
Insertionsort	9	9	2	9	9	9	9	9	9	9	0	9	9	9	9	9	9	9	16.0
Member	1	1	0	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1.7
Mergesort	9	9	0	9	1	9	2	9	4	9	0	9	9	9	3	9	7	9	14.1
Permutations	7	7	2	7	7	7	7	7	7	7	0	7	7	7	7	7	7	7	12.4
QuicksortA	8	8	0	6	2	3	1	6	5	8	0	8	8	8	8	8	8	8	14.3
QuicksortC	8	8	0	8	1	7	1	8	3	8	0	8	6	8	5	8	6	8	13.4
Reverse	12	12	1	12	6	11	6	12	11	12	0	12	11	12	3	12	11	12	20.9
ReverseAcc	8	8	0	8	6	7	7	8	7	8	0	8	8	8	8	8	7	8	15.6
ReverseRev	2	2	0	2	0	0	0	0	2	2	0	2	2	2	0	2	2	2	3.8
Selectsort	9	9	2	9	7	8	7	9	8	9	0	9	8	9	8	9	8	9	14.2
Treesort	6	6	0	6	1	6	1	6	4	6	0	6	5	6	1	6	5	6	10.2
			E_1	E_2	E_3	E_4	E_5	E_6	E_7	E_8	Z_1	Z_2	Z_3	Z_4	Z_5	Z_6	Z_7	Z_8	
Total	114	114	9	110	59	99	59	109	87	114	0	114	109	114	83	114	104	114	202.5

transformation (not shown in Table 1) are usually quite small. In particular, for the 109 properties proved *sat* by Z3, the verification time was almost always below 1s. Only for 13 of them, it was between 1s and 4s. For the remaining five *sat* properties, Z3 exceeded the timeout limit.

Out of the 114 *sat* properties, Eldarica proved 9 *sat* properties (all relative to list size) before transformation and 59 *sat* properties (relative also to catamorphisms different from list size) after transformation (see columns E_1 and E_3). However, one property that was proved *sat* before transformation, was not proved *sat* after transformation. This is the only example where the built-in *size* function of Eldarica has been more effective than our transformation-based approach.

Given the 114 *unsat* properties, Z3 proved all of them to be *unsat* before transformation and also after transformation (see columns Z_2 and Z_4). The proofs before transformation took well-below 1s in almost all examples, and after transformation took an equal or shorter time for more than half of the cases.

Given the 114 *unsat* properties, Eldarica proved 110 of them to be *unsat* before transformation, and only 99 of them after transformation (see columns E_2 and E_4). This is the only case where we experienced a degradation of performance after transformation. This degradation may be related to the facts that: (i) the number of clauses in the transformed set $T_{w\&wo}$ is larger than the number of clauses in the source set, and (ii) the clauses in $T_{w\&wo}$ have often more atoms in their bodies with respect to the source clauses.

If we consider the set T_w, instead of $T_{w\&wo}$, we have a significant decrease in the number of clauses and the number of atoms in the bodies of clauses. In this case, Z3 proved 83 properties to be *sat* (less than for $T_{w\&wo}$, see columns Z_3 and Z_5) and all 114 properties to be *unsat* (as for all other input sets of clauses, see columns Z_2, Z_4, and Z_6). Eldarica proved 59 properties to be *sat* (the same as for $T_{w\&wo}$, see columns E_3 and E_5) and 109 properties to be *unsat* (almost the same as for the source clauses, see columns E_2 and E_6).

Finally, we have considered the set T_{wo}, instead of $T_{w\&wo}$. For the 114 *sat* properties, Eldarica proved 87 of them (see column E_7), while Z3 proved 104 of them (see column Z_7). For the *unsat* properties both Eldarica and Z3 proved all of them (see columns E_8 and Z_8). However, since T_{wo} computes an overapproximation with respect to $T_{w\&wo}$ (and also with respect to T_w), when the solver returns the answer *unsat*, one cannot conclude that the property at hand is indeed *unsat*. Both solvers, in fact, wrongly classified 10 *sat* properties as *unsat*.

In summary, our experimental evaluation shows that VeriCaT$_{abs}$ with Z3 as back-end solver outperforms the other CHC solving tools we have considered. Indeed, our tool shows much higher effectiveness than the others when verifying *sat* properties, while it retains the excellent performance of Z3 for *unsat* properties.

6 Conclusions and related work

It is well known that the proof of many program properties can be reduced to a proof of satisfiability of sets of CHCs (Bjørner et al., 2015; De Angelis et al., 2022; Gurfinkel, 2022). In order to make it easier to automatically prove satisfiability, whenever a program is made out of many functions, possibly recursively defined and depending on each other,

https://doi.org/10.1017/S147106842400019X Published online by Cambridge University Press

it is commonly suggested to provide properties also for the auxiliary functions that may occur in the program. Those extra properties basically play the role of lemmas, which often make the proof of a property of interest much easier.

We have focused our study on the automatic proof of properties of programs that compute over ADTs, when these properties can be defined using catamorphisms. In a previous paper (De Angelis et al., 2023), we have proposed an algorithm for dealing with a multiplicity of properties of the various program functions to be proved at the same time. In this paper, we have investigated an approach, whereby the auxiliary properties need not be explicitly defined, but it is enough to indicate the catamorphisms involved in their specifications. This leaves to the CHC solver the burden of discovering the suitable auxiliary properties needed for the proof of the property of interest. Thus, this much simpler requirement we make avoids the task of providing all the properties of the auxiliary functions occurring in the program. However, in principle, the proofs of the properties may become harder for the CHC solver. Our experimental evaluation shows that this is not the case if we follow a transformation-based approach. Indeed, the results presented in this paper support the following two-step approach: (1) use algorithm $\mathcal{T}_{abs}$ proposed here to derive a new, transformed set of CHCs from the given initial set of CHCs that translate the program together with its property of interest, and then, (2) use the Z3 solver with *global guidance* (Krishnan et al., 2020) on the derived set.

We have shown that our approach is a valid alternative to the development of algorithms for extending CHC solvers with special purpose mechanisms that handle ADTs. In fact, recently proposed approaches extend CHC solvers to the case of CHCs over ADTs through the use of various mechanisms such as: (i) the combination with inductive theorem proving (Unno et al., 2017), (ii) the lemma generation based on syntax-guided synthesis from user-specified templates (Yang et al., 2019), (iii) the invariant discovery based on finite tree automata (Kostyukov et al., 2021), and (iv) the use of suitable abstractions on CHCs with recursively defined function symbols (Govind V. K., Shoham, and Gurfinkel, 2022).

One key feature of our algorithm $\mathcal{T}_{abs}$ is that it is sound and complete with respect to satisfiability, that is, the transformed set of CHCs is satisfiable if and only if so is the initial one. In this respect, our results here improve over previous work (De Angelis et al., 2022), where algorithm $\mathcal{T}_{cata}$ only preserves soundness, that is, if the transformed set of CHCs is satisfiable, then so is the initial one, while if the transformed set is unsatisfiable, nothing can be inferred for the given set.

In our experiments, we have also realized the usefulness of having more catamorphisms acting together when verifying a specific property. For instance, in the case of the quicksort program, when using the catamorphism *is_asorted* alone, Z3 is unable to show (within the timeout of 600s) sortedness of the output list, while when using also the catamorphisms *listmin* and *listmax*, after transformation Z3 proved sortedness in less than 2s. We leave it for future work to automatically derive the catamorphisms that are useful for showing the property of interest, even if they are not strictly necessary for specifying that property.

Our approach is very much related to *abstract interpretation* (Cousot and Cousot, 1977), which is a methodology for checking properties by interpreting the program as computing over a given abstract domain. Catamorphisms can be seen as specific abstraction functions. Abstract interpretation techniques have been studied also in the field of logic programming. In particular, the CiaoPP preprocessor (Hermenegildo et al., 2005) implements abstract interpretation techniques that use *type-based norms*, which are a special kind of integer-valued catamorphisms. These techniques have important applications in *termination analysis* (Bruynooghe et al., 2007) and *resource analysis* (Albert et al., 2020).

Usually, abstract interpretation is the basis for sound analysis techniques by computing an (over-)approximation of the concrete semantics of a program, and hence these techniques may find counterexamples to the properties of interest that hold in the abstract semantics, but that are not feasible in the concrete semantics. As already mentioned, our transformation guarantees the equisatisfiability of the initial and the transformed CHCs, and hence all counterexamples found are feasible in the initial CHCs.

Among the various abstract interpretation techniques, the one which is most related to our verification approach, is the so-called *model-based abstract interpretation* (Gallagher et al., 1995). This abstract interpretation technique is based on the idea of defining a *pre-interpretation*, that is, an interpretation of the function symbols of a logic program over a specified domain of interest. That pre-interpretation is used for generating, via *abstract compilation* (De Angelis et al., 2022, Sec. 4.3), a *domain program* whose least model is an abstraction of the least model of the original program. Then, program properties can be inferred from the model of the domain program. One similarity is that pre-interpretations of ADT constructors can be seen as catamorphisms. Actually, our definition of a catamorphism is more general than the one of a pre-interpretation, in that: (i) we admit non-ADT additional parameters as, for instance, in the *listcount* predicate of our introductory example, and (ii) we allow mutually dependent predicates in the definitions of catamorphisms. Another similarity is that the abstract compilation used by model-based abstract interpretation can be seen as a program transformation and, indeed, it can be implemented by partial evaluation. However, as already mentioned for other abstract interpretation techniques, that transformation does not guarantee equisatisfiability and by using it, one can prove the satisfiability of the original set of clauses, but not its unsatisfiability.

Our transformation-based approach is, to a large extent, parametric with respect to the theory of constraints used in the CHCs. Thus, it can easily be extended to theories different from *LIA* and *Bool* used in this paper, and in particular, to other theories such as linear real/rational arithmetic or bit-vectors, as far as they are supported by the CHC solver. This is a potential advantage with respect to those abstract interpretation techniques that require the design of an ad-hoc abstract domain for each specific program analysis.

Acknowledgments

We thank Arie Gurfinkel for helpful suggestions on the use of the Z3 (SPACER) solver. We also thank John Gallagher and the anonymous referees of LOPSTR 2023 for helpful comments on previous versions of the paper. Finally, we express our gratitude to Robert

Glück and Bishoksan Kafle for inviting us to write this improved, extended version of our LOPSTR 2023 paper. The authors are members of the INdAM Research Group GNCS.

Competing interests

The authors declare none.

References

Albert, E., Genaim, S., Gutiérrez, R. and Martin-Martin, E. 2020. A transformational approach to resource analysis with typed-norms inference. *Theory and Practice of Logic Programming* 20, 3, 310–357.

Barrett, C. W., Sebastiani, R., Seshia, S. A. and Tinelli, C. 2009. Satisfiability modulo theories. In *Handbook of Satisfiability* vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 825–885.

Bjørner, N., Gurfinkel, A., McMillan, K. L. and Rybalchenko, A. 2015. Horn clause solvers for program verification. In *Fields of Logic and Computation (II)*, vol. 9300. Lecture Notes in Computer Science. Springer, 24–51.

Blicha, M., Fedyukovich, G., Hyvärinen, A. E. J. and Sharygina, N. 2022. Transition power abstractions for deep counterexample detection, In *Tools and Algorithms for the Construction and Analysis of Systems TACAS '22, Part I*, vol. 13243. Lecture Notes in Computer Science. Springer, 524–542.

Bruynooghe, M., Codish, M., Gallagher, J. P., Genaim, S. and Vanhoof, W. 2007. Termination analysis of logic programs through combination of type-based norms. *ACM Transactions on Programming Languages and Systems* 29, 2, 10–es.

Cousot, P. and Cousot, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *POPL'77: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Los Angeles, California, USA, 238–252.

De Angelis, E., Fioravanti, F., Gallagher, J. P., Hermenegildo, M. V., Pettorossi, A. and Proietti, M. 2022. Analysis and transformation of constrained Horn clauses for program verification. *Theory and Practice of Logic Programming* 22, 6, 974–1042.

De Angelis, E., Fioravanti, F., Pettorossi, A. and Proietti, M. 2014. VeriMAP: A tool for verifying programs through transformations. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS '14*, vol. 8413. Lecture Notes in Computer Science. Springer, 568–574.

De Angelis, E., Fioravanti, F., Pettorossi, A. and Proietti, M. 2022. Satisfiability of constrained Horn clauses on algebraic data types: A transformation-based approach. *Journal of Logic and Computation* 32, 2, 402–442.

De Angelis, E., Fioravanti, F., Pettorossi, A. and Proietti, M. 2023. Multiple query satisfiability of constrained Horn clauses. In *Practical Aspects of Declarative Languages*, Hanus, M. and Inclezan, D., Eds., vol. 13880, Lecture Notes in Computer Science. Springer, 125–143.

De Angelis, E. and Govind, V. K. 2022. CHC-COMP 2022: Competition report. In *Proc. 9th Workshop on Horn Clauses for Verification and Synthesis and 10th Int. Workshop on Verification and Program Transformation. EPTCS*, vol. 373, Munich, Germany, Open Publishing Association, 44–62.

De Angelis, E., Proietti, M., Fioravanti, F. and Pettorossi, A. 2022. Verifying catamorphism-based contracts using constrained Horn clauses. *Theory and Practice of Logic Programming* 22, 4, 555–572.

de Moura, L. M. and Bjørner, N. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS '08*, vol. 4963. Lecture Notes in Computer Science. Springer, 337–340.

Etalle, S. and Gabbrielli, M. 1996. Transformations of CLP modules. *Theoretical Computer Science* 166, 1-2, 101–146.

Gallagher, J. P., Boulanger, D. and Saglam, H. 1995. Practical model-based static analysis for definite logic programs. In *International Symposium on Logic Programming*, Lloyd, J. W. Ed. MIT Press, 351–365.

Govind, V. K. H., Shoham, S. and Gurfinkel, A. 2022. Solving constrained Horn clauses modulo algebraic data types and recursive functions. In *Proc. of the ACM on Programming Languages, POPL'22*, Philadelphia, PA, USA, vol. 6, 1–29.

Gurfinkel, A. 2022. Program verification with constrained Horn clauses (invited paper). In *34th Computer Aided Verification*, Shoham, S. and Vizel, Y., Eds., vol. 13371. Lecture Notes in Computer Science. Springer, 19–29.

Hermenegildo, M. V., Puebla, G., Bueno, F. and López-García, P. 2005. Integrated program debugging, verification, and optimization using abstract interpretation (and the Ciao system preprocessor). *Science of Computer Programming* 58, 1-2, 115–140.

Hinze, R., Wu, N. and Gibbons, J. 2013. Unifying structured recursion schemes. In *International Conference on Functional Programming, ICFP '13. ACM*, 209–220.

Hojjat, H. and Rümmer, P. 2018. The ELDARICA Horn solver. In *Formal Methods in Computer Aided Design, FMCAD '18*. IEEE, 1–7.

Jaffar, J. and Maher, M. 1994. Constraint logic programming: A survey. *Journal of Logic Programming* 19, 20, 503–581.

Komuravelli, A., Gurfinkel, A. and Chaki, S. 2016. SMT-based model checking for recursive programs. *Formal Methods in System Design* 48, 3, 175–205.

Kostyukov, Y., Mordvinov, D. and Fedyukovich, G. 2021. Beyond the elementary representations of program invariants over algebraic data types. In *Conference on Programming Language Design and Implementation, PLDI '21*. ACM, 451–465.

Krishnan, H. G. V., Chen, Y., Shoham, S. and Gurfinkel, A. 2020. Global guidance for local generalization in model checking. In *CAV '20, Part II*. Lahiri, S. K. and Wang, C., Eds., vol. 12225. Lecture Notes in Computer Science. Springer, 101–125.

Lloyd, J. W. 1987. *Foundations of Logic Programming*. 2nd ed. Springer-Verlag, Berlin.

Meijer, E., Fokkinga, M. M. and Paterson, R. 1991. Functional programming with bananas, lenses, envelopes and barbed wire. In *5th ACM Conference on Functional Programming Languages and Computer Architecture*, vol. 523. Lecture Notes in Computer Science. Springer, 124–144.

Suter, P., Köksal, A. S. and Kuncak, V. 2011. *Satisfiability modulo recursive programs. Symposium on Static Analysis, SAS '11*, vol. 6887. Lecture Notes in Computer Science. Springer, 298–315.

Tamaki, H. and Sato, T. 1986. A generalized correctness proof of the unfold/fold logic program transformation. Technical Report 86-4. Ibaraki University, Japan.

Unno, H., Torii, S. and Sakamoto, H. 2017. Automating induction for solving Horn clauses. In *29th CAV '17, Part II*, vol. 10427. Lecture Notes in Computer Science. Springer, 571–591.

Yang, W., Fedyukovich, G. and Gupta, A. 2019. Lemma synthesis for automating induction over algebraic data types. In *International Conference on Principles and Practice of Constraint Programming*, vol. 11802. Lecture Notes in Computer Science. Springer, 600–617.