TLP **18** (3–4): 452–469, 2018. © Cambridge University Press 2018
doi:10.1017/S1471068418000157

Solving Horn Clauses on Inductive Data Types Without Induction

EMANUELE DE ANGELIS and FABIO FIORAVANTI
DEC, 'G. d'Annunzio' University of Chieti-Pescara, Pescara, Italy
(*e-mails:* {emanuele.deangelis, fabio.fioravanti}@unich.it)

ALBERTO PETTOROSSI
DICII, University of Rome Tor Vergata, Rome, Italy
(*e-mail:* pettorossi@info.uniroma2.it)

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

submitted 27 April 2018; accepted 11 May 2018

Abstract

We address the problem of verifying the satisfiability of Constrained Horn Clauses (CHCs) based on theories of inductively defined data structures, such as lists and trees. We propose a transformation technique whose objective is the removal of these data structures from CHCs, hence reducing their satisfiability to a satisfiability problem for CHCs on integers and booleans. We propose a transformation algorithm and identify a class of clauses where it always succeeds. We also consider an extension of that algorithm, which combines clause transformation with reasoning on integer constraints. Via an experimental evaluation we show that our technique greatly improves the effectiveness of applying the Z3 solver to CHCs. We also show that our verification technique based on CHC transformation followed by CHC solving, is competitive with respect to CHC solvers extended with induction.

KEYWORDS: Program verification, constrained Horn clauses, constraint logic programming, inductively defined data types, program transformation.

1 Introduction

Constraint logic programs have become a well-established formalism for solving program verification problems (Albert *et al.* 2007, Bjørner *et al.* 2015, De Angelis *et al.* 2014, Jaffar *et al.* 2012, Méndez-Lojo *et al.* 2008, Peralta *et al.* 1998). In the verification field, constraint logic programs are often called constrained Horn clauses (CHCs), and here we will adopt this terminology. The verification method based on CHCs consists in reducing a program verification problem to the satisfiability of a set of CHCs. Since CHC satisfiability, also called CHC *solving*, is in general an undecidable problem, some heuristics have been proposed in the literature, such as *Counterexample Guided Abstraction Refinement* (CEGAR) (Clarke *et al.* 2000), *Craig interpolation* (McMillan 2003), and *Property Directed Reachability* (PDR) (Bradley 2011, Hoder and Bjørner 2012). Some tools, called CHC solvers, for verifying the satisfiability of CHCs are available. We

https://doi.org/10.1017/S1471068418000157 Published online by Cambridge University Press

recall Eldarica (Hojjat *et al.* 2012), HSF (Grebenshchikov *et al.* 2012), RAHFT (Kafle *et al.* 2016), VeriMAP (De Angelis *et al.* 2014), and Z3 (de Moura and Bjørner 2008). CHC solvers make use of a combination of the above heuristics and have been shown to be very effective for CHCs with several underlying constraint theories, such as the theory of Linear Integer Arithmetic (*LIA*) and the theory of Boolean constraints (*Bool*).

Solving techniques have also been developed for CHCs manipulating inductively defined data structures such as lists and trees (see, for instance, `https://rise4fun.com/Z3/tutorial/guide` for the Z3 solver). However, CHC solvers acting on those data structures are usually less effective than CHC solvers for clauses with constraints in *LIA* or *Bool*. This is mainly due to the fact that methods for satisfiability used by CHC solvers are based on variants of resolution, augmented with ad hoc algorithms for the underlying constraint theory, and no induction principles for the data structures are used.

To overcome this difficulty, recent work has proposed the extension of CHC solving by adding a principle of induction on predicate derivations (Unno *et al.* 2017). This work is on the same line of other proposals which extend techniques for Satisfiability Modulo Theory (SMT) with inductive reasoning (Reynolds and Kuncak 2015, Suter *et al.* 2011) by incorporating methods derived from the field of automated theorem proving.

In this paper we propose an alternative method to solve CHCs on inductively defined data structures. It is based on the application of suitable transformations of CHCs that have the objective of removing those data structures while preserving satisfiability of clauses. Our transformation method makes use of the fold/unfold transformation rules (Etalle and Gabbrielli 1996, Tamaki and Sato 1984), and it is based on some techniques proposed in the past for improving the efficiency of functional and logic programs, such as *deforestation* (Wadler 1990), *unnecessary variable elimination* (Proietti and Pettorossi 1995), and *conjunctive partial deduction* (De Schreye *et al.* 1999). However, the focus of the method presented in this paper is on the improvement of effectiveness of CHC solvers, rather than the improvement of efficiency of programs.

In our method we separate the concern of reasoning on inductively defined data structures from the concern of proving clause satisfiability. This separation of concerns eases the task of the CHC solvers in many cases. For instance, when the constraints are on trees of integers, our transformation method, if successful, allows us to derive an equisatisfiable set of clauses with constraints on integers only.

The main contributions of our work are the following. (1) We propose a transformation algorithm, called $\mathcal{E}$, that removes inductively defined data structures from CHCs and derives new, equisatisfiable sets of clauses whose constraints are from the *LIA* theory and the *Bool* theory (see Section 4). (2) We identify a class of CHCs where Algorithm $\mathcal{E}$ is guaranteed to terminate, and all inductively defined data structures can be removed (see Section 5). (3) By using reasoning techniques on *LIA* constraints, we derive an extension of our transformation algorithm $\mathcal{E}$, called Algorithm $\mathcal{EC}$ (see Section 6). (4) We report on an experimental evaluation of a tool that implements Algorithm $\mathcal{EC}$. We consider CHCs that encode properties of OCaml functional programs, and we show that our tool is competitive with the RCaml tool that extends the Z3 solver by adding induction rules for reasoning on inductive data structures (Unno *et al.* 2017) (see Section 7).

https://doi.org/10.1017/S1471068418000157 Published online by Cambridge University Press

2 A Tree Processing Example

As an introductory example to illustrate our technique for proving program properties without using induction on data structures, let us consider the following *Tree_Processing* program, which we write according to the OCaml syntax (Leroy *et al.* 2017).

```
type tree = Leaf ——— Node of int * tree * tree ;;
let min x y = if x<y then x else y ;;
let rec min_leaf t = match t with
                   | Leaf ->0
                   | Node(x,l,r) ->1+min (min_leaf l) (min_leaf r) ;;
let rec left_drop n t = match t with
                   | Leaf -> Leaf
                   | Node(x,l,r) ->  if n<=0 then Node(x,l,r) else left_drop (n−1) l ;;
```

Let us also consider the following non-trivial property *Prop* to be verified for the *Tree_Processing* program:

$$\forall n\ \forall t.\ n \geq 0\ \Rightarrow\ \big(((\mathit{min\_leaf}\ (\mathit{left\_drop}\ n\ t)) + n) \geq (\mathit{min\_leaf}\ t)\big).$$

Now we translate the *Tree_Processing* program and the property *Prop* into a set of CHCs that are satisfiable iff *Prop* holds for *Tree_Processing*. We get the following set of clauses (clauses 1–7 for the program and clause 8 for the property):

1. $\mathit{min}(X,Y,Z) \leftarrow X<Y,\ Z=X$
2. $\mathit{min}(X,Y,Z) \leftarrow X \geq Y,\ Z=Y$
3. $\mathit{min\_leaf}(\mathit{leaf},M) \leftarrow M=0$
4. $\mathit{min\_leaf}(\mathit{node}(X,L,R),M) \leftarrow M=M3+1,\ \mathit{min\_leaf}(L,M1),\ \mathit{min\_leaf}(R,M2),$
   $\mathit{min}(M1,M2,M3)$
5. $\mathit{left\_drop}(N,\mathit{leaf},\mathit{leaf}) \leftarrow$
6. $\mathit{left\_drop}(N,\mathit{node}(X,L,R),\mathit{node}(X,L,R)) \leftarrow N \leq 0$
7. $\mathit{left\_drop}(N,\mathit{node}(X,L,R),T) \leftarrow N \geq 1,\ N1=N-1,\ \mathit{left\_drop}(N1,L,T)$
8. $\mathit{false} \leftarrow N \geq 0,\ M+N<K,\ \mathit{left\_drop}(N,T,U),\ \mathit{min\_leaf}(U,M),\ \mathit{min\_leaf}(T,K)$

In clause 8 the head is *false* because the relation '<' in the constraint $M+N<K$ occurring in the body is the negation of the relation '≥' occurring in the property *Prop*.

The Z3 solver is *not* able to prove the satisfiability of clauses 1–8 (which amounts to show that *false* is not derivable), because of the presence of variables of type *tree*. One can extend the capabilities of Z3 by adding induction rules on trees as done in recent work (Reynolds and Kuncak 2015, Suter *et al.* 2011, Unno *et al.* 2017). Instead, as we propose in this paper, we can derive by transformation, starting from clauses 1–8, a new, equisatisfiable set of clauses without variables of type *tree* and whose constraints are in *LIA* only, and then we can use CHC solvers, such as Z3, to show the satisfiability of this new set of clauses. Starting from clauses 1–8, our transformation algorithm, whose details are given in Section 4, produces the following clauses, where *min* is defined by clauses 1 and 2:

9. $\mathit{new1}(N,M,K) \leftarrow M=0,\ K=0$
10. $\mathit{new1}(N,M,K) \leftarrow N \leq 0,\ M=M3+1,\ K=M,\ \mathit{new2}(M1),\ \mathit{new2}(M2),$
    $\mathit{min}(M1,M2,M3)$
11. $\mathit{new1}(N,M,K) \leftarrow N \geq 1, N1=N-1, K=K3+1,\ \mathit{new1}(N1,M,K1),\ \mathit{new2}(K2),$
    $\mathit{min}(K1,K2,K3)$

https://doi.org/10.1017/S1471068418000157 Published online by Cambridge University Press

12. $new2(M) \leftarrow M = 0$
13. $new2(M) \leftarrow M = M3+1,\ new2(M1),\ new2(M2),\ min(M1, M2, M3)$
14. $false \leftarrow N \geq 0,\ M+N < K,\ new1(N, M, K)$

Now, the Z3 solver for CHCs with *LIA* constraints is able to prove the satisfiability of clauses $1, 2, 9$–14 without using any induction on trees. The details of the transformation from clauses 1–8 to clauses $1, 2, 9$–14 will be presented in Section 4.

3 Preliminaries

Let us consider a typed, first order functional language. We assume that the *basic types* of the language are *int*, for integers, and *bool*, for booleans. We also have *non-basic types*, which are introduced by (possibly recursive) type definitions such as the one for trees of integers considered in Section 2. The type system of our language can be formally defined as follows. (One may consider richer type systems including, for instance, parameterized types, but this system is sufficient for presenting our verification technique.)

$$\begin{array}{l} \mathit{Types} \ni \tau ::= \mathit{int} \ \mid\ \mathit{bool} \ \mid\ \mathit{ident} \ \mid\ \tau_1 * \ldots * \tau_k \\ \mathit{Type\ Definitions}\colon \mathit{ident} = c_1 \ \mathsf{of}\ \tau_1 \ \mid\ \ldots \ \mid\ c_n \ \mathsf{of}\ \tau_n \end{array}$$

where: (i) *ident* is a type identifier, (ii) the operator '$*$' builds k-tuple types, (iii) $c_1, \ldots, c_n$ are distinct constructors with arity, and (iv) in any expression 'c_i of τ_i', if c_i has arity $k > 0$, then τ_i is a k-tuple type, and if c_i has arity 0, then '$\mathsf{of}\ \tau_i$' is absent.

Every function has a *functional type* of the form $\tau_1 \rightarrow \tau_2$, for some types τ_1 and τ_2, modulo the usual isomorphism between $\tau \rightarrow (\tau' \rightarrow \tau'')$ and $(\tau * \tau') \rightarrow \tau''$.

In the following definitions: (i) n is an integer, (ii) b is *true* or *false*, (iii) x is a typed variable, (iv) c is a constructor of arity k (≥ 0), and (v) f is a function of arity k (≥ 0) or a primitive operator on integers or booleans such as: $+$, $\times$, $=$, $\neq$, $\leq$, $\neg$, $\wedge$, and $\Rightarrow$.

$$\begin{array}{l} \mathit{Values} \ni v ::= n \ \mid\ b \ \mid\ c\ v_1 \ldots v_k \\ \mathit{Terms} \ni t ::= n \ \mid\ b \ \mid\ x \ \mid\ c\ t_1 \ldots t_k \ \mid\ f\ t_1 \ldots t_k \ \mid\ \mathsf{if}\ t_0\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 \\ \qquad\qquad \mid\ \mathsf{let}\ x = t_0\ \mathsf{in}\ t_1 \ \mid\ \mathsf{match}\ x\ \mathsf{with}\ \mid p_1\ \text{-¿}\ t_1 \ \mid\ \ldots \ \mid p_n\ \text{-¿}\ t_n \\ \mathit{Patterns} \ni p ::= c(x_1, \ldots, x_k) \end{array}$$

We assume that all values, terms, and patterns are well-typed. In every let-in term x is a new variable not occurring in t_0 and occurring in t_1. In every match-with term the patterns $p_1, \ldots, p_k$ are pairwise disjoint and exhaustive. A *user-defined function* f is defined by: $\mathsf{let\ rec}\ f\ x_1 \ldots x_k = t$, where the free variables of t are among $\{x_1, \ldots, x_k\}$.

A *program* is a (possibly empty) set of type definitions together with a set of user-defined functions. Given a program P and a term t without free variables and whose functions are defined in P, the value v of t using P is computed according to the call-by-value semantics. In this case we write $t \rightarrow_P v$. We say that a program P *terminates* if, for every term t without free variables and whose functions are defined in P, there exists a value v such that $t \rightarrow_P v$. Given a boolean term q whose free variables are in $\{x_1, \ldots, x_n\}$, a *property* is a universally quantified formula of the form $\forall x_1, \ldots, x_n . q$. Given a program P, we say that a property $\forall x_1, \ldots, x_n . q$, whose functions are defined in P, holds for P iff for all values $v_1, \ldots, v_n$, we have that $q[v_1, \ldots, v_n \,/\, x_1, \ldots, x_n] \rightarrow_P \mathit{true}$, where the square bracket notation is for substitution.

https://doi.org/10.1017/S1471068418000157 Published online by Cambridge University Press

Let us consider a typed first order logic (Enderton 1972) which includes: (i) the theory *LIA* of the linear integer arithmetic constraints, and (ii) the theory *Bool* of boolean constraints. A *constraint* in $LIA \cup Bool$ is any formula in $LIA \cup Bool$.

An *atom* is a formula of the form $q(t_1, \ldots, t_m)$, where q is a typed predicate symbol not used in $LIA \cup Bool$, and $t_1, \ldots, t_m$ are typed terms made out of variables and constructors. A *constrained Horn clause* (or simply, a *clause*, or a CHC) is an implication of the form $A \leftarrow c, B$ (comma denotes conjunction), where the conclusion (or *head*) A is either an atom or *false*, the premise (or *body*) is the conjunction of a constraint c, and a (possibly empty) conjunction B of atoms. A clause whose head is an atom is called a *definite clause*, and a clause whose head is *false* is called a *goal*. A *constrained fact* is a clause of the form $A \leftarrow c$. We will write the constrained fact $A \leftarrow true$ also as $A \leftarrow$. We assume that all variables in a clause are universally quantified in front. Given a term t, by $vars(t)$ we denote the set of variables occurring in t. Similarly for the set of variables occurring in atoms, or clauses, or sets of atoms, or sets of clauses. A set S of CHCs is said to be *satisfiable* if $S \cup LIA \cup Bool$ has a model, or equivalently, $S \cup LIA \cup Bool \not\models false$.

We define a translation Tr from any set P of function definitions into a set of CHCs by: (i) introducing for each function f of arity k, a new predicate, say p, of arity $k{+}1$, and then (ii) providing the clauses for predicate p so that $f(x_1, \ldots, x_k){=}y$ iff $p(x_1, \ldots, x_k, y)$ holds. This translation has been applied in Section 2 for generating clauses 1–7 starting from the *Tree_Processing* program. Primitive operations, such as $+$ and $-$, are translated in terms of constraints expressed in *LIA*. Similarly, a property of the form: $\forall x.\, r(x) \Rightarrow s(x)$ is translated into a goal (or a set of goals, in general) as indicated in Section 2 (see goal 8). If we denote by $Tr(P)$ and $Tr(Prop)$ the set of CHCs generated from a given program P and a given property *Prop*, respectively, we have the following theorem, where satisfiability is defined within the typed logic we consider.

Theorem 1
Given a program P that terminates, a property *Prop* holds for P iff the set $Tr(P) \cup Tr(Prop)$ of clauses is satisfiable.

4 Eliminating Inductively Defined Data Structures

Now we present an algorithm, called Algorithm $\mathcal{E}$, for eliminating from CHCs the predicate arguments that have a non-basic type, such as lists or trees. If Algorithm $\mathcal{E}$ terminates, then it transforms a set of clauses into an equisatisfiable set, where the arguments of all predicates have basic types. The algorithm is based on the fold/unfold strategy for eliminating unnecessary variables from logic programs (Proietti and Pettorossi 1995).

Given two terms t_1 and t_2, by $t_1 \preceq t_2$ and $t_1 \prec t_2$ we denote the *subterm* and *strict subterm* relation, respectively. Given an atom A, by $nbargs(A)$ we denote the set of arguments of non-basic type of A. By $nbvars(A)$ we denote the set of variables of non-basic type occurring in A. The next definition extends the $\preceq$ and $\prec$ relations to atoms.

Definition 1 (*Atom Comparison*)
Given two atoms A_1 and A_2, $A_1 \preceq A_2$ (or $A_1 \prec A_2$) holds if there exist $t_1 \in nbargs(A_1)$ and $t_2 \in nbargs(A_2)$ such that: (i) $vars(t_1) \cap vars(t_2) \neq \emptyset$, and (ii) $t_1 \preceq t_2$ (or $t_1 \prec t_2$). Given a set S of atoms, an atom $M \in S$ is *strictly maximal* in S if there exists no atom $A \in S$ such that $M \prec A$ and there exists an atom $A' \in S$ such that $A' \prec M$.

https://doi.org/10.1017/S1471068418000157 Published online by Cambridge University Press

For example, $q([Y|Ys],[\,])$ is strictly maximal in the set $\{p([X],Ys),\ q([Y|Ys],[\,])\}$. Indeed, $Ys \prec [Y|Ys]$, and hence $p([X],Ys) \prec q([Y|Ys],[\,])$, while $q([Y|Ys],[\,]) \not\prec p([X],Ys)$ (note that $[\,] \prec [X]$, but $vars([\,]) \cap vars([X]) = \emptyset$). The notion of a strictly maximal atom will be used in the *Unfold* procedure of Algorithm $\mathcal{E}$ to guide the unfolding process.

A predicate *has basic types* if *all* its arguments have basic type. An atom has basic types if its predicate has basic types. A clause has basic types if *all* its atoms have basic types.

Definition 2 (*Sharing Blocks*)
Let S be a set (or a conjunction) of atoms. For any two atoms A_1 and A_2 in S, $A_1 \downarrow A_2$ holds iff $nbvars(A_1) \cap nbvars(A_2) \neq \emptyset$. Let $\Downarrow$ be the reflexive transitive closure of $\downarrow$. By $SharingBlocks(S)$ we denote the partition of S into subsets, called *sharing blocks*, with respect to $\Downarrow$.

Algorithm $\mathcal{E}$ makes use of the well-known transformation rules *define*, *fold*, *unfold*, and *replace* for CHCs (Etalle and Gabbrielli 1996, Tamaki and Sato 1984).

The Elimination Algorithm $\mathcal{E}$.
Input: A set $Cls \cup Gs$, where Cls is a set of definite clauses and Gs is a set of goals;
Output: A set *TransfCls* of clauses such that: (1) $Cls \cup Gs$ is satisfiable iff *TransfCls* is satisfiable, and (2) every clause in *TransfCls* has basic types.

$Defs := \emptyset$; $InCls := Gs$; $TransfCls := \emptyset$;
while $InCls \neq \emptyset$ **do**
  Define-Fold($Defs, InCls, NewDefs, FldCls$);
  Unfold($NewDefs, Cls, UnfCls$);
  Replace($UnfCls, Cls, RCls$);
  $Defs := Defs \cup NewDefs$; $InCls := RCls$; $TransfCls := TransfCls \cup FldCls$;

Starting from a set *Cls* of definite clauses and a set *Gs* of goals, Algorithm $\mathcal{E}$ applies iteratively the procedures *Define-Fold*, *Unfold*, and *Replace*, in this order, until it derives a set *TransfCls* of clauses whose predicates have basic types only. Algorithm $\mathcal{E}$ collects in *Defs* the clauses, called *definitions*, introduced by the *Define* steps, and collects in *InCls* the clauses to be transformed. The set *Defs* of definitions is initialized to the empty set and the set *InCls* of the input goals is initialized to the set *Gs* of goals.

Now let us present the various procedures used by Algorithm $\mathcal{E}$.

Procedure *Define-Fold*($Defs, InCls, NewDefs, FldCls$)
Input: A set *Defs* of definitions and a set *InCls* of clauses;
Output: A set *NewDefs* of definitions and a set *FldCls* of clauses.

$NewDefs := \emptyset$; $FldCls := \emptyset$;
for each clause C: $H \leftarrow c, B$ in *InCls* **do**
  if C is a constrained fact **then** $FldCls := FldCls \cup \{C\}$ **else**
    Define. Let $SharingBlocks(B) = \{B_1, \ldots, B_n\}$;
      for $i = 1, \ldots, n$ **do**
      if there is no clause in $Defs \cup NewDefs$ whose body is B_i (modulo the names of variables and the order and multiplicity of the atoms) **then**

https://doi.org/10.1017/S1471068418000157 Published online by Cambridge University Press

NewDefs := *NewDefs* ∪ {$newp_i(V_i) \leftarrow B_i$}
where: (i) $newp_i$ is a new predicate symbol, and (ii) V_i is the tuple of distinct variables of basic type occurring in B_i;

Fold. C is folded using the definitions in *Defs* ∪ *NewDefs*, thereby deriving
F: $H \leftarrow c, newp_1(V_1), \ldots, newp_n(V_n)$
where, for $i = 1, \ldots, n$, $newp_i(V_i) \leftarrow B_i$ is the unique clause in *Defs* ∪ *NewDefs* whose body is B_i, modulo variable renaming;
FldCls := *FldCls* ∪ {F};

The *Define-Fold* procedure removes the arguments with non-basic types from each clause C of *InCls* in two steps: first, the *Define* step introduces a new predicate definition for each sharing block of the body of C (unless a definition with a body equal, modulo variable renaming, to that block has already been introduced in a previous *Define* step), and then, the *Fold* step replaces each sharing block by the head of the new definition. Since the heads of these new definitions have basic types, the body of the clause of the form: $H \leftarrow c, newp_1(V_1), \ldots, newp_n(V_n)$, derived from C by the *Fold* step, has basic types. Also H has basic types, because: (i) Algorithm $\mathcal{E}$ initializes *InCls* to the set *Gs* of goals, whose heads are *false*, and (ii) the head predicate of each new clause added to *InCls* has been introduced by a previous *Define* step, and hence, by construction, has basic types.

Example 1
(*Tree_Processing*). Let us consider the introductory example of Section 2. At the first iteration of the while-do body of Algorithm $\mathcal{E}$, *InCls* consists of clause 8, which is the result of translating the property *Prop* to be verified. The *Define* step introduces the new predicate *new*1 through the following definition:

15. $new1(N, M, K) \leftarrow left\_drop(N, T, U),\ min\_leaf(U, M),\ min\_leaf(T, K)$

The body of this clause consists of the single sharing block of the body of clause 8 (note that $left\_drop(N, T, U)$ shares the tree variables T and U with $min\_leaf(T, K)$ and $min\_leaf(U, M)$, respectively). Now, the *Fold* step derives clause 14 of Section 2, where all predicates have arguments of integer type, and thus it is added to the final set *TransfCls*.

The *Define* step adds to *Defs* a set *NewDefs* of new definitions, whose body may have predicates with non-basic types (see clause 15). Now, Algorithm $\mathcal{E}$ proceeds by applying the *Unfold* procedure to those clauses.

Procedure *Unfold*(*NewDefs*, *Cls*, *UnfCls*)
Input: A set *NewDefs* of definitions and a set *Cls* of definite clauses;
Output: A set *UnfCls* of clauses.

UnfCls := *NewDefs*;
Initially, all atoms in the body of the clauses of *UnfCls* are marked 'unfoldable';
while there exists a clause C in *UnfCls* of the form $H \leftarrow c, L, A, R$ such that either
(i) the atom A is 'unfoldable' and is strictly maximal in L, A, R, or
(ii) all atoms in L, A, R are 'unfoldable' and not strictly maximal **do**
Let $K_1 \leftarrow c_1, B_1, \ \ldots, \ K_m \leftarrow c_m, B_m$ be all clauses of *Cls* (where, without loss of generality, we assume $vars(Cls) \cap vars(C) = \emptyset$) such that, for $i = 1, \ldots, m$, (1) there exists a most general unifier ϑ_i of A and K_i, and (2) the constraint $(c, c_i)\vartheta_i$ is satisfiable;

https://doi.org/10.1017/S1471068418000157 Published online by Cambridge University Press

$$UnfCls := (UnfCls - \{C\}) \cup \{(H \leftarrow c, c_1, L, B_1, R)\vartheta_1, \ldots, (H \leftarrow c, c_m, L, B_m, R)\vartheta_m\}$$

where, for $i = 1, \ldots, m$, (1) an atom $E\vartheta_i$ of $(L, R)\vartheta_i$ is 'unfoldable' iff the corresponding atom E of (L, R) is 'unfoldable' in C, and (2) no atom in $B\vartheta_i$ is 'unfoldable';

The *Unfold* procedure unfolds the atoms occurring in *NewDefs* by performing resolution steps with clauses in *Cls*. The procedure applies a strategy that consists in unfolding strictly maximal atoms, if any. In the case where the predicates are defined by induction on the structure of their arguments with non-basic types, this strategy corresponds to a form of induction on the arguments structure. The subsequent folding steps correspond to applications of the inductive hypotheses. A characterization of a class of CHCs where this strategy is successful will be given in Section 5. The use of the 'unfoldable' marking on atoms enforces a finite number of resolution steps.

Note that when Case (ii) of the condition of the while-do holds, the *Unfold* procedure may unfold *any* 'unfoldable' atom (and in our implementation of the procedure we unfold the leftmost one). However, in the class of CHCs presented in Section 5, the termination of Algorithm $\mathcal{E}$ is independent of the choice of the atom to be unfolded.

Example 2

(*Tree_Processing, continued*). Let us continue the derivation presented in Example 1. All atoms in the body of clause 15 are 'unfoldable' and none is strictly maximal (indeed, no tree argument is a strict superterm of another). In this case the *Unfold* procedure may unfold any 'unfoldable' atom, and we assume that it unfolds $left\_drop(N, T, U)$. The following three clauses, where we have underlined the 'unfoldable' atoms, are derived:

16. $new1(N, M, K) \leftarrow \underline{min\_leaf(leaf, M)},\ \underline{min\_leaf(leaf, K)}$
17. $new1(N, M, K) \leftarrow N \leq 0,\ \underline{min\_leaf(node(X, L, R), M)},\ \underline{min\_leaf(node(X, L, R), K)}$
18. $new1(N, M, K) \leftarrow N \geq 1,\ N1 = N - 1,\ left\_drop(N1, L, U),\ \underline{min\_leaf(U, M)},$
    $\underline{min\_leaf(node(X, L, R), K)}$

Now the *Unfold* procedure continues by selecting more atoms for unfolding. For instance, in clause 18 it selects $min\_leaf(node(X, L, R), K)$, which is strictly maximal because the argument L of $left\_drop(N1, L, U)$ is a strict subterm of $node(X, L, R)$ and no atom in the body of clause 18 has an argument that is a strict superterm of $node(X, L, R)$. After five iterations, where all underlined atoms, except $min\_leaf(U, M)$, are unfolded, the *Unfold* procedure derives the following clauses:

19. $new1(N, M, K) \leftarrow M = 0,\ K = 0$
20. $new1(N, M, K) \leftarrow N \leq 0,\ M = M3 + 1,\ K = K3 + 1,$
    $min\_leaf(L, M1),\ min\_leaf(R, M2),$
    $min(M1, M2, M3),\ min\_leaf(L, K1),\ min\_leaf(R, K2),\ min(K1, K2, K3)$
21. $new1(N, M, K) \leftarrow N \geq 1,\ N1 = N - 1\ K = K3 + 1,\ left\_drop(N1, L, U),$
    $\underline{min\_leaf(U, M)},$
    $\underline{min\_leaf(L, K1)},\ min\_leaf(R, K2),\ min(K1, K2, K3)$

In clause 21, $min\_leaf(U, M)$ is an 'unfoldable' atom, but the *Unfold* procedure stops because that atom is not strictly maximal. Clause 19 is a constrained fact and it is added to the final set *TransfCls* (indeed, clause 19 is clause 9 of Section 2).

https://doi.org/10.1017/S1471068418000157 Published online by Cambridge University Press

After the *Unfold* procedure Algorithm $\mathcal{E}$ may simplify some clauses by exploiting functional dependencies among predicate arguments, which hold by construction for the predicates obtained by translating functional programs.

Definition 3 (*Predicate Functionality*)
Let *Cls* be a set of definite clauses. A predicate $p(X,Y)$, where X and Y are tuples of arguments, is *functional* in *Cls* if $Cls \cup \{false \leftarrow Y \neq Z,\ p(X,Y),\ p(X,Z)\}$ is a satisfiable set of clauses.

Procedure *Replace*(*UnfCls*, *Cls*, *RCls*)
Input: A set *UnfCls* of clauses and a set *Cls* of definite clauses;
Output: A set *RCls* of clauses.

RCls := *UnfCls*;
while there is a clause $C \in RCls$ of the form: $H \leftarrow c, G_1, p(t,u), G_2, p(t,w), G_3$, where predicate $p(X,Y)$ is functional in *Cls* **do**
Replace C by $(H \leftarrow c, G_1, p(t,u), G_2, G_3)\vartheta$, where ϑ is a most general unifier of u and w;

Example 3
(*Tree_Processing*, *continued*). By the functionality of the predicates $min\_leaf(T,M)$ and $min(X,Y,Z)$, clause 20 is replaced by the following one:

22. $new1(N,M,K) \leftarrow N \leq 0, M{=}M3{+}1, K{=}M,\ min\_leaf(L,M1),\ min\_leaf(R,M2),$
$min(M1,M2,M3)$

Now, Algorithm $\mathcal{E}$ performs a second iteration by executing again the *Define-Fold*, *Unfold*, and *Replace* procedures. The *Define* step introduces the following definition:

23. $new2(M1) \leftarrow min\_leaf(L,M1)$

whose body consists of a sharing block in the body of clause 22 (note that the tree variable L is not shared with any other atom). No other definitions are introduced, as all other sharing blocks in the clauses currently in *InCls* (namely, clauses 19, 21, and 22) are variants of the body of the definitions 15 and 23. The *Fold* step derives clauses 10 and 11 of Section 2 from clauses 22 and 21, respectively.

Finally, Algorithm $\mathcal{E}$ performs an *Unfold* step followed by a *Fold* step on clause 23 and derives clauses 12 and 13 of Section 2. Note that *Replace* steps are not applicable, and no new definitions are introduced by the *Define* step. Thus, Algorithm $\mathcal{E}$ terminates and returns clauses $1, 2, 9$–14 of Section 2.

5 Correctness and Termination of the Transformation

The partial correctness of Algorithm $\mathcal{E}$ follows from well-known satisfiability preservation results that hold for the fold/unfold transformation rules (Etalle and Gabbrielli 1996, Tamaki and Sato 1984) (for a proof, see the online appendix corresponding to this paper at the TPLP archives).

https://doi.org/10.1017/S1471068418000157 Published online by Cambridge University Press

Theorem 2 (*Partial Correctness*)
Let *Cls* be a set of definite clauses and let *Gs* be a set of goals. If Algorithm $\mathcal{E}$ terminates for the input clauses $Cls \cup Gs$, returning a set *TransfCls* of clauses, then (1) $Cls \cup Gs$ is satisfiable iff *TransfCls* is satisfiable, and (2) all clauses in *TransfCls* have basic types.

Now we introduce a class of CHCs where Algorithm $\mathcal{E}$ is guaranteed to terminate.

First we need the following terminology and notation. An atom is said to be *linear* if each variable occurs in it at most once. By $arity(p)$ we denote the arity of predicate symbol p.

Definition 4 (*Slice Decomposition*)
Let C be a clause of the form $A_0 \leftarrow c, A_1, \ldots, A_m$, and let *Pred* be the set of predicate symbols in the atoms $A_0, A_1, \ldots, A_m$ of C. A *slice* of C is a total function $\sigma : Pred \rightarrow \mathbb{N}$ such that, for every predicate $p \in Pred$: (1) $0 \leq \sigma(p) \leq arity(p)$, and (2) if $\sigma(p) = i > 0$, then the i-th argument of p has a non-basic type. For any atom $p(t_1, \ldots, t_k)$ in C, we define:

$$\sigma(p(t_1, \ldots, t_k)) = \begin{cases} p_0 & \text{if } \sigma(p) = 0 \\ p_i(t_i) & \text{if } \sigma(p) = i > 0 \end{cases}$$

where p_0 (of arity 0) and p_i (of arity 1) are fresh, new predicate symbols.

We also define $\sigma(C)$ to be the new clause $\sigma(A_0) \leftarrow \sigma(A_1), \ldots, \sigma(A_m)$. The slice σ of C is *quasi-descending* if $\sigma(A_0), \sigma(A_1), \ldots, \sigma(A_m)$ are linear atoms and, for $i = 1, \ldots, m$, (1) for $j = 1, \ldots, m$, $vars(\sigma(A_i)) \cap vars(\sigma(A_j)) = \emptyset$, if $i \neq j$, and (2) either $vars(\sigma(A_0)) \cap vars(\sigma(A_i)) = \emptyset$ or $\sigma(A_0) \succeq \sigma(A_i)$.

A *slice decomposition* of clause C is a set $\Sigma_C = \{\sigma_1, \ldots, \sigma_n\}$, where $\sigma_1, \ldots, \sigma_n$ are slices of C and, for all $p \in Pred$, for all $i \in \{1, \ldots, arity(p)\}$, if the i-th argument of p has non-basic type, then there exists $\sigma_j \in \Sigma_C$ such that $\sigma_j(p) = i$. A slice decomposition Σ_C of a clause C is said to be *disjoint* if, for any two slices σ_h and σ_k in Σ_C, we have that $vars(\sigma_h(C)) \cap vars(\sigma_k(C)) = \emptyset$, whenever $h \neq k$. Σ_C is said to be *quasi-descending* if all slices in Σ_C are quasi-descending.

The following are slices of clauses 4 and 7 (see Section 2):

$\sigma_1(\text{clause } 4) = min\_leaf_1(node(X, L, R)) \leftarrow min\_leaf_1(L), min\_leaf_1(R), min_0$
$\sigma_2(\text{clause } 7) = left\_drop_2(node(X, L, R)) \leftarrow left\_drop_2(L)$
$\sigma_3(\text{clause } 7) = left\_drop_3(T) \leftarrow left\_drop_3(T)$

We have that $\Sigma_4 = \{\sigma_1\}$ is a disjoint, quasi-descending slice decomposition of clause 4, and so is $\Sigma_7 = \{\sigma_2, \sigma_3\}$ for clause 7.

Now we define a class of goals for which Algorithm $\mathcal{E}$ terminates.

Definition 5 (*Sharing Cycle*)
Let G be a goal of the form $false \leftarrow c, A_1, \ldots, A_m$. We say that G has a *sharing cycle* if there is a sequence of $k\,(>1)$ distinct variables $X_0, \ldots, X_{k-1}$ of non-basic type, and a sequence $A'_0, \ldots, A'_k$ of atoms in $A_1, \ldots, A_m$, such that: (i) $A'_0 = A'_k$, (ii) $A'_1, \ldots, A'_k$ are all distinct, and (iii) for $i = 0, \ldots, k-1$, A'_i and A'_{i+1} share the non-basic variable X_i.

Now, if we represent the body of goal 8 of Section 2 as the following labeled graph:

$$min\_leaf(T, K) \xrightarrow{\quad T \quad} left\_drop(N, T, U) \xrightarrow{\quad U \quad} min\_leaf(U, M)$$

where the arc from a node A to a node B has label X iff $X \in nbvars(A) \cap nbvars(B)$, then it is easy to see that goal 8 has no sharing cycle (indeed, there are no cycles in the above graph).

https://doi.org/10.1017/S1471068418000157 Published online by Cambridge University Press

We have the following result, whose proof is given in the online appendix corresponding to this paper at the TPLP archives.

Theorem 3 (*Termination*)
Let *Cls* be a set of definite clauses such that every clause in *Cls* has a disjoint, quasi-descending slice decomposition. Let *Gs* be a set of goals such that, for each goal $G \in Gs$, (i) G is of the form $false \leftarrow c, A_1, \ldots, A_m$, where for $i = 1, \ldots, m$, A_i is an atom whose arguments are distinct variables, and (ii) G has no sharing cycles. Then Algorithm $\mathcal{E}$ terminates for the input clauses $Cls \cup Gs$.

Note that all definite clauses (clauses 1–7) of our introductory example have a quasi-descending slice decomposition (see above for clauses 4 and 7). These slice decompositions are also disjoint, except for the one of clause 6 where the second and third argument of *left_drop* share some variables. However, clause 6 can be rewritten as:

$left\_drop(N, node(X, L, R), node(X1, L1, R1)) \leftarrow N \leq 0,\ X{=}X1,\ eqt(L, L1),\ eqt(R, R1)$
$eqt(leaf, leaf) \leftarrow$
$eqt(node(X1, L1, R1), node(X2, L2, R2)) \leftarrow X1 = X2,\ eqt(L1, L2),\ eqt(R1, R2)$

where predicate *eqt* defines the equality between binary trees. These three clauses have a disjoint, quasi-descending slice decomposition, and after this rewriting the termination of Algorithm $\mathcal{E}$ is guaranteed. The rewriting of any constrained fact, such as clause 6, into a clause that has a disjoint, quasi-descending slice decomposition can be done automatically as a pre-processing step, by introducing an equality predicate for each non-basic type. However, in the benchmark set presented in Section 7, this pre-processing step has no effect on the termination behavior of our transformation algorithms.

6 Adding Integer and Boolean Constraints

Algorithm $\mathcal{E}$ is not guaranteed to terminate outside the class of definite clauses and goals considered in Theorem 3. Let us consider, for instance, the following set of clauses which specifies a verification problem on lists:

$append([\,], Ys, Ys) \leftarrow$
$append([X|Xs], Ys, [Z|Zs]) \leftarrow X = Z,$
$\quad append(Xs, Ys, Zs)$

$drop(N, [\,], [\,]) \leftarrow$
$drop(N, [X|Xs], [Y|Xs]) \leftarrow N = 0, X = Y$
$drop(N, [X|Xs], Ys) \leftarrow N \neq 0, N1 = N - 1,$
$\quad drop(N1, Xs, Ys)$

$take(N, [\,], [\,]) \leftarrow$
$take(N, [X|Xs], [\,]) \leftarrow N = 0$
$take(N, [X|Xs], [Y|Ys]) \leftarrow N \neq 0,\ X = Y,$
$\quad N1 = N - 1,\ take(N1, Xs, Ys)$
$diff\_list([\,], [Y|Ys]) \leftarrow$
$diff\_list([X|Xs], [\,]) \leftarrow$
$diff\_list([X|Xs], [Y|Ys]) \leftarrow X \neq Y$
$diff\_list([X|Xs], [Y|Ys]) \leftarrow X = Y,$
$\quad diff\_list(Xs, Ys)$

$false \leftarrow M = N,\ take(M, Xs, Ys), drop(N, Xs, Zs), append(Ys, Zs, A), diff\_list(A, Xs)$

In these clauses: (i) $take(M, Xs, Ys)$ holds if the list Ys is the prefix of the list Xs up to its M-th element, (ii) $drop(N, Xs, Zs)$ holds if list Zs is a suffix of Xs starting from its $(N{+}1)$-th element, (iii) $append(Ys, Zs, A)$ holds if A is the concatenation of the lists Ys and Zs, and (iv) $diff\_list(A, Xs)$ holds if A and Xs are different lists.

https://doi.org/10.1017/S1471068418000157 Published online by Cambridge University Press

The definite clauses listed above satisfy the hypothesis of Theorem 3 (after rewriting some constrained facts as done at the end of the previous section), but the goal does not. Indeed, that goal has the following sharing cycle:

$$take(M,Xs,Ys) \xrightarrow{Xs} drop(N,Xs,Zs) \xrightarrow{Zs} append(Ys,Zs,A) \xrightarrow{Ys} take(M,Xs,Ys)$$

Algorithm $\mathcal{E}$ does not terminate on this example. Indeed, starting from the definition:

$$new0(M,N) \leftarrow take(M,Xs,Ys), drop(N,Xs,Zs), append(Ys,Zs,A), \ diff\_list(A,Xs)$$

infinitely many new predicates with unbounded lists are generated by Algorithm $\mathcal{E}$. However, these predicates correspond to cases where $M{\neq}N$, and if we keep the constraint $M{=}N$ between the first arguments of *take* and *drop*, then a finite set of new predicates is generated. In particular, if we start from the definition:

$$new1(M,N) \leftarrow M{=}N, take(M,Xs,Ys), drop(N,Xs,Zs), append(Ys,Zs,A), \\ diff\_list(A,Xs)$$

the transformation terminates after a few steps and derives the following equisatisfiable set of clauses, where an equality constraint holds for the two arguments of each occurrence of *new*1:

$$new2 \leftarrow new2$$
$$new1(M,N) \leftarrow M{=}N, \ M{=}0, \ new2$$
$$new1(M,N) \leftarrow M{=}N, \ M{\neq}0, \ M{=}1{+}M1, \ N{=}1{+}N1, \ new1(M1,N1)$$
$$false \leftarrow M{=}N, \ new1(M,N)$$

The satisfiability of this set of clauses is trivial, because it does not contain any constrained fact, and is easily proved by the Z3 solver.

The above example motivates the introduction of a variant of Algorithm $\mathcal{E}$, called Algorithm $\mathcal{EC}$, which is obtained by allowing in the *Define* step the introduction of definitions whose bodies may include constraints in $LIA \cup Bool$, and hence applying *Fold* steps with respect to these definitions. The rest of Algorithm $\mathcal{EC}$ is equal to Algorithm $\mathcal{E}$.

As usual in constraint-based transformation techniques (see (De Angelis *et al.* 2017b) for a recent paper), the computation of a suitable constraint when introducing a new definition is done by means of a *constraint generalization* function *Gen*. We say that a constraint g is *more general than*, or it is a *generalization of*, a constraint c, if $LIA \cup Bool \models \forall(c \rightarrow g)$. Given a constraint c, a conjunction B of atoms, and a set *Defs* of definitions, the function *Gen* matches the constraint c against the constraint d occurring in a definition in *Defs* whose body is of the form d, B (modulo variable renaming), and returns a new constraint $Gen(c, B, Defs)$ which is more general than both c and d.

The details of how the constraint generalization function *Gen* is actually implemented are not necessary for understanding Algorithm $\mathcal{EC}$, which is parametric with respect to such a function. Let us only mention here that the function *Gen* used for the experiments reported in Section 7 makes use on the *widening operator* based on *bounded difference shapes*, which is a standard operator on convex polyhedra considered in the field of *abstract interpretation* (Bagnara *et al.* 2008, Cousot and Halbwachs 1978). The use of widening avoids the introduction of infinitely many definitions that differ for the constraints only.

https://doi.org/10.1017/S1471068418000157 Published online by Cambridge University Press

The *Define* and *Fold* steps for Algorithm $\mathcal{EC}$ are as follows.

Let C: $H \leftarrow c, B$ in *InCls* be a clause which is not a constrained fact.

Define. Let $SharingBlocks(B) = \{B_1, \ldots, B_n\}$;

for $i = 1, \ldots, n$ **do**

$g_i := Gen(c, B_i, Defs \cup NewDefs)$;

if there is no clause in $Defs \cup NewDefs$ whose body is g_i, B_i

then $NewDefs := NewDefs \cup \{newp_i(V_i) \leftarrow g_i, B_i\}$

where: (i) $newp_i$ is a new predicate symbol, and (ii) V_i is the tuple of distinct variables of basic type occurring in B_i;

Fold. C is folded using the definitions in $Defs \cup NewDefs$, thereby deriving

F: $H \leftarrow c, newp_1(V_1), \ldots, newp_n(V_n)$

where, for $i = 1, \ldots, n$, $newp_i(V_i) \leftarrow g_i, B_i$ is the unique clause in $Defs \cup NewDefs$ whose body is g_i, B_i, modulo variable renaming;

$FldCls := FldCls \cup \{F\}$;

The partial correctness of Algorithm $\mathcal{E}$ (see Theorem 2) carries over to Algorithm $\mathcal{EC}$, because also $\mathcal{EC}$ can be expressed as a sequence of applications of the fold/unfold transformation rules for CHCs (Etalle and Gabbrielli 1996). Also the termination result for Algorithm $\mathcal{E}$ (see Theorem 3) extends to $\mathcal{EC}$, as long as the function *Gen* guarantees that, for a given conjunction B of atoms, finitely many definitions of the form $newp(V) \leftarrow g, B$ can be introduced.

7 Experimental Evaluation

In this section we present the results of an experimental evaluation we have performed for assessing the effectiveness of our approach and, in particular, for comparing it with the approach that extends CHC solvers by adding inductive rules, as done in the RCaml tool (Unno *et al.* 2017) based on the Z3 solver.

Implementation. We have implemented the transformation strategy presented in Section 4 using the VeriMAP system (De Angelis *et al.* 2014) together with the Parma Polyhedra Library (PPL) (Bagnara *et al.* 2008) for performing constraint generalizations. Then, we have used the Z3 solver v4.6.0 with the Spacer fixed-point engine (Komuravelli *et al.* 2013) to check the satisfiability of the transformed CHCs.

The tool and the benchmark suite are available at `https://fmlab.unich.it/iclp2018/`.

Benchmark suite and experiments. Our benchmark suite is a collection of 105 verification problems, each one consisting of an OCaml functional program manipulating inductively defined data structures (such as lists or trees) together with a property to be verified. Most of the problems (70 out of 105) derive from the benchmark suite of RCaml (see `http://www.cs.tsukuba.ac.jp/ uhiro/`, Software *RCaml*, [web demo (induction)]). This suite, in turn, includes problems from the suite of IsaPlanner (Dixon and Fleuriot 2003) which is a generic framework for proof planning in the Isabelle theorem prover. We divide our benchmark suite into four sets of problems (see Table 1): (1) *FirstOrder*, the set of problems relative to first-order programs (57 out of 70) in the RCaml suite, (2) *HigherOrderInstances*, the set of problems relative to first-order programs (13 out of

https://doi.org/10.1017/S1471068418000157 Published online by Cambridge University Press

Table 1. *Column n reports the number of problems in each Problem Set. Columns* S_{Z3} *and* T_{Z3} *report the number of problems which have been solved by Z3 and the total time needed for their solution. Analogously, for the two columns referring to* $\mathcal{EC}$;Z3 *and the two columns referring to* RCaml. *Times are in seconds. The timeout occurs after 300s.*

		Z3		$\mathcal{EC}$;Z3		RCaml	
Problem Set	n	S_{Z3}	T_{Z3}	$S_{\mathcal{EC};\text{Z3}}$	$T_{\mathcal{EC};\text{Z3}}$	S_{RCaml}	T_{RCaml}
(1) *FirstOrder*	57	3	0.09	47	37.64	41	216.59
(2) *HigherOrderInstances*	13	1	0.04	11	8.33	10	45.40
(3) *MoreLists*	16	3	13.87	14	11.27	10	119.01
(4) *MoreTrees*	19	5	20.18	19	26.79	5	55.16
Total	105	12	34.18	91	84.03	66	436.17
Avg time			2.85		0.92		6.61

70) that have been obtained by instantiating higher-order programs in the RCaml suite, (3) *MoreLists*, a set of 16 verification problems on lists, and (4) *MoreTrees*, a set of 19 verification problems on trees. In our benchmark 94 programs do satisfy the associated property and the remaining 11 do not.

By using the preprocessor provided by the RCaml system, each verification problem has been translated into a set of CHCs (see the translation *Tr* of Section 3). Then, for each derived set, call it I, of CHCs we have performed the following three experiments, whose results are summarized in Table 1. (A table with the detailed results for each problem of the benchmark is available at `https://fmlab.unich.it/iclp2018/`.)

- We have run Z3 (which does not use any structural induction rule) for checking the satisfiability of I (see the two columns for Z3).
- We have applied the transformation algorithm $\mathcal{EC}$ to I, thereby producing a set T of CHCs, and then we have run Z3 for checking the satisfiability of T (see the two columns for $\mathcal{EC}$;Z3).
- We have run RCaml on I (see the two columns for RCaml).

For each verification problem we have set a timeout limit of 300 seconds. Z3 and VeriMAP have run on an Intel Xeon CPU E5-2640 2.00GHz with 64GB under CentOS. RCaml has run in a Linux virtual machine on an Intel i5 2.3GHz with 8GB memory under macOS.

Results. The figures in Table 1 show that our approach considerably increases the effectiveness of CHC satisfiability checking. Indeed, when directly applied to the CHCs that encode the given verification problems, Z3 is able to solve 12 problems, while it solves 91 problems when it is applied to the CHCs produced by our transformation algorithm $\mathcal{EC}$ (compare Columns S_{Z3} and $S_{\mathcal{EC};\text{Z3}}$). For the remaining 14 problems, Algorithm $\mathcal{EC}$ does not terminate within the time limit, and thus no CHCs are produced. Note that these 14 problems fall outside the class of programs for which Algorithm $\mathcal{E}$, and also Algorithm $\mathcal{EC}$, is guaranteed to terminate.

Table 1 shows that our approach compares quite well with respect to the induction based approach implemented in the RCaml system (Unno *et al.* 2017). Indeed, $\mathcal{EC}$;Z3 proves 65 out of 66 problems that are also proved by RCaml, and in addition it proves 26 problems, 7 of which belong to the RCaml benchmark suite. Also the average times

https://doi.org/10.1017/S1471068418000157 Published online by Cambridge University Press

appear to be favorable to $\mathcal{EC}$;Z3 with respect to RCaml. In particular, on the set of 67 problems where both $\mathcal{EC}$;Z3 and RCaml provide the solution within the timeout, $\mathcal{EC}$;Z3 is about six times faster than RCaml, having taken into account the fact that the machine on which we have run RCaml is approximately 13% slower.

Now let us report on some other important facts not shown in Table 1. When solving verification problems via $\mathcal{EC}$;Z3, a substantial portion of work is performed by Algorithm $\mathcal{EC}$. Indeed, the total time spent by Z3 for solving the 91 problems is only 6% of the total solving time (84.03s). Moreover, for 42 problems the set of clauses produced by $\mathcal{EC}$ does not contain constrained facts, and thus its satisfiability can immediately be checked by Z3.

Unfortunately, it may happen that our transformation prevents the solution of some verification problems. Indeed, the transformation algorithm $\mathcal{EC}$ does not terminate within the timeout on two problems that can be proved using Z3 alone. However, on the 10 problems that are solved by both Z3 and $\mathcal{EC}$;Z3, the average time taken by $\mathcal{EC}$;Z3 (3.41s) to solve one problem is much lower than that taken by Z3 alone (9.74s).

Finally, let us comment on our benchmark suite. Many of the problems are small, but the properties to be verified are not trivial, such as those presented in Sections 2 and 6. One of the most difficult problems solved by $\mathcal{EC}$;Z3, but not by RCaml, consists in showing that the insertion of a node in a binary search tree produces again a binary search tree. That solution took about 5s. The solution of some of the 13 problems on which both $\mathcal{EC}$;Z3 and RCaml run out of time, requires the use of suitable lemmas. For instance, one of these problems can be proved by using a lemma stating that the sum of the elements of the concatenation of two lists is equal to the sum of the elements of the first list plus the sum of the elements of the second list. The extension of our approach to support automatic lemma discovery is left for future work.

8 Related Work and Conclusions

We have presented a two-step approach to the verification of programs that manipulate inductively defined data structures, such as lists and trees. The first step consists in the transformation of the set of clauses that encodes the given verification problem, into a new, equisatisfiable set of clauses whose variables do no longer refer to inductive data structures. The second step consists in the application of a CHC solver to the derived set of clauses with integer and boolean constraints only. Thus, in our approach we can take full advantage of the many efficient solvers that exist for clauses with integer and boolean constraints (de Moura and Bjørner 2008, Grebenshchikov *et al.* 2012, Hoder and Bjørner 2012, Hojjat *et al.* 2012, Komuravelli *et al.* 2013). Through some experiments we have shown that, using an implementation of our algorithm and the Z3 solver, our two-step approach is competitive with respect to other techniques that extend CHC solvers by adding deduction rules for reasoning on inductive data structures (Unno *et al.* 2017).

Our transformation technique is related to methods proposed in the past for improving the efficiency of functional and logic programs, such as *deforestation* (Wadler 1990), *unnecessary variable elimination* (UVE) (Proietti and Pettorossi 1995), and *conjunctive partial deduction* (De Schreye *et al.* 1999). Among these techniques, the UVE transformation, which makes use of the fold/unfold rules, is the most similar to the one presented

https://doi.org/10.1017/S1471068418000157 Published online by Cambridge University Press

in this paper. However, in this paper we have introduced several technical novelties with respect to UVE: (i) the use of type information, (ii) the use of constraints, and (iii) a better characterization of the termination of the main algorithm (in particular, here we have introduced the notions of a slice decomposition and a circular sharing). At a more general level, the objective of the work presented in this paper is to show the usefulness of our transformation techniques for the improvement of the effectiveness of CHC solvers, rather than the improvement of the execution of logic programs.

The idea of using a transformation-based approach for the verification of software comes from the area of Constraint Logic Programming, where program specialization has been applied as a means of deriving CLP programs from interpreters of imperative languages (Albert *et al.* 2007, De Angelis *et al.* 2017c, Méndez-Lojo *et al.* 2008, Peralta *et al.* 1998). CHC solvers based on combinations of transformations and abstract interpretation have also been developed (De Angelis *et al.* 2014, Kafle *et al.* 2016) and have been shown to be competitive with solvers based on CEGAR, Interpolation, and PDR.

Recently, CHCs have been proposed for verifying *relational* program properties (Felsing *et al.* 2014), that is, properties that relate two programs, such as equivalence. It has also been shown that *predicate pairing*, which is a fold/unfold transformation for CHCs, greatly improves the effectiveness of CHC solvers for relational verification (De Angelis *et al.* 2015, De Angelis *et al.* 2016, De Angelis *et al.* 2017a). A related CHC transformation technique, called *CHC product*, works by composing pairs of clauses with an effect similar to predicate pairing, although in some cases it may derive sets of clauses with fewer predicates (Mordvinov and Fedyukovich 2017). Neither predicate pairing nor CHC product can remove inductively defined data structures, as done by the transformation technique presented in this paper.

Similarly to the technique presented in this paper, fold/unfold transformations and constraint generalization have also been used in a verification technique for imperative programs that compute over arrays (De Angelis *et al.* 2017b). However, the above mentioned technique is not able to remove array data structures, and unlike the one presented here, it does not consider inductively defined data structures.

We plan to extend our transformation-based verification method in a few directions, and in particular we plan: (i) to study the problem of automatically generating the lemmas which are sometimes needed for removing data structures, and (ii) to consider the verification problem for higher-order functional programs.

9 Acknowledgements

We would like to thank Hiroshi Unno for his support in the use of the RCaml system. We thank the anonymous reviewers for their constructive comments. The authors are members of the INdAM Research group GNCS. E. De Angelis, F. Fioravanti and A. Pettorossi are research associates at CNR-IASI, Rome, Italy.

Supplementary material

To view supplementary material for this article, please visit http://dx.doi.org/10.1017/S1471068418000157

https://doi.org/10.1017/S1471068418000157 Published online by Cambridge University Press

References

Albert, E., Gómez-Zamalloa, M., Hubert, L., and Puebla, G. 2007. Verification of Java bytecode using analysis and transformation of logic programs. In *Proc. of PADL '07*, LNCS 4354. Springer, 124–139.

Bagnara, R., Hill, P. M., and Zaffanella, E. 2008. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.

Bjørner, N., Gurfinkel, A., McMillan, K. L., and Rybalchenko, A. 2015. Horn clause solvers for program verification. In *Fields of Logic and Computation II*, LNCS 9300. Springer, 24–51.

Bradley, A. R. 2011. SAT-based model checking without unrolling. In *Proc. of VMCAI '11*, LNCS 6538. Springer, 70–87.

Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., and Veith, H. 2000. Counterexample-Guided Abstraction Refinement. In *Proc. of CAV '00*, LNCS 1855. Springer, 154–169.

Cousot, P. and Halbwachs, N. 1978. Automatic discovery of linear restraints among variables of a program. In *Proc. of POPL '78*. ACM, 84–96.

De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. 2014. VeriMAP: A tool for verifying programs through transformations. In *Proc. of TACAS '14*, LNCS 8413. Springer, 568–574.

De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. 2015. Proving correctness of imperative programs by linearizing constrained Horn clauses. *Theory and Practice of Logic Programming 15*, 635–650.

De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. 2016. Relational verification through Horn clause transformation. In *Proc. of SAS '16*, LNCS 9837. Springer, 147–169.

De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. 2017a. Predicate pairing for program verification. *Theory and Practice of Logic Programming*, 1–41. Published online, to appear in press (`https://arxiv.org/abs/1708.01473`).

De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. 2017b. Program Verification using Constraint Handling Rules and array constraint generalizations. *Fundamenta Informaticae 150*, 73–117.

De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. 2017c. Semantics-based generation of verification conditions via program specialization. *Science of Computer Programming 147*, 78–108.

de Moura, L. M. and Bjørner, N. 2008. Z3: An efficient SMT solver. In *Proc. of TACAS '08*. LNCS 4963. Springer, 337–340.

De Schreye, D., Glück, R., Jørgensen, J., Leuschel, M., Martens, B., and Sørensen, M. H. 1999. Conjunctive partial deduction: Foundations, control, algorithms, and experiments. *Journal of Logic Programming 41,* 2–3, 231–277.

Dixon, L. and Fleuriot, J. D. 2003. IsaPlanner: A prototype proof planner in Isabelle. In *Proc. of CADE-19*, LNCS 2741. Springer, 279–283.

Enderton, H. 1972. *A Mathematical Introduction to Logic*. Academic Press.

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

Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., and Ulbrich, M. 2014. Automating regression verification. In *Proc. of ACM/IEEE Conf. ASE '14*. 349–360.

Grebenshchikov, S., Lopes, N. P., Popeea, C., and Rybalchenko, A. 2012. Synthesizing software verifiers from proof rules. In *Proc. of ACM SIGPLAN PLDI '12*. 405–416.

Hoder, K. and Bjørner, N. 2012. Generalized property directed reachability. In *Proc. of SAT '12*, LNCS 7317. Springer, 157–171.

https://doi.org/10.1017/S1471068418000157 Published online by Cambridge University Press

Hojjat, H., Konecný, F., Garnier, F., Iosif, R., Kuncak, V., and Rümmer, P. 2012. A verification toolkit for numerical transition systems. In *Proc. of FM '12*, LNCS 7436. Springer, 247–251.

Jaffar, J., Navas, J. A., and Santosa, A. E. 2012. Unbounded symbolic execution for program verification. In *Proc. of RV '11*. LNCS 7186. Springer, 396–411.

Kafle, B., Gallagher, J. P., and Morales, J. F. 2016. RAHFT: A tool for verifying Horn clauses using abstract interpretation and finite tree automata. In *Proc. of CAV '16*. LNCS 9779. Springer, 261–268.

Komuravelli, A., Gurfinkel, A., Chaki, S., and Clarke, E. M. 2013. Automatic abstraction in SMT-based unbounded software model checking. In *Proc. of CAV '13*, LNCS 8044. Springer, 846–862.

Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., and Vouillon, J. 2017. The OCaml system, Release 4.06. INRIA, France.

McMillan, K. L. 2003. Interpolation and SAT-based model checking. In *Proc. of CAV '03*. LNCS 2725. Springer, 1–13.

Méndez-Lojo, M., Navas, J. A., and Hermenegildo, M. V. 2008. A flexible, (C)LP-based approach to the analysis of object-oriented programs. In *Proc. of LOPSTR'07*. LNCS 4915. Springer, 154–168.

Mordvinov, D. and Fedyukovich, G. 2017. Synchronizing constrained Horn clauses. In *Proc. of LPAR-21 EPiC Series in Computing Vol. 46*. EasyChair, 338–355.

Peralta, J. C., Gallagher, J. P., and Saglam, H. 1998. Analysis of imperative programs through analysis of constraint logic programs. In *Proc. of SAS '98*, LNCS 1503. Springer, 246–261.

Proietti, M. and Pettorossi, A. 1995. Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs. *Theoretical Computer Science 142,* 1, 89–124.

Reynolds, A. and Kuncak, V. 2015. Induction for SMT solvers. In *Proc. of VMCAI '15*, LNCS 8931. Springer, 80–98.

Suter, P., Köksal, A. S., and Kuncak, V. 2011. Satisfiability modulo recursive programs. In *Proc. of SAS '11*, LNCS 6887. Springer, 298–315.

Tamaki, H. and Sato, T. 1984. Unfold/fold transformation of logic programs. In *Proc. of ICLP '84*, S.-Å. Tärnlund, Ed., Uppsala University, Sweden, 127–138.

Unno, H., Torii, S., and Sakamoto, H. 2017. Automating induction for solving Horn clauses. In *Proc. of CAV '17*. LNCS 10427. Springer, 571–591.

Wadler, P. L. 1990. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science 73*, 231–248.

https://doi.org/10.1017/S1471068418000157 Published online by Cambridge University Press