A Coinduction Rule for Entailment of Recursively Defined Properties

Joxan Jaffar, Andrew E. Santosa, and Răzvan Voicu

School of Computing
National University of Singapore
Republic of Singapore 117543
{joxan,andrews,razvan}@comp.nus.edu.sg

Abstract. Recursively defined properties are ubiquitous. We present a proof method for establishing entailment $\mathcal{G} \models \mathcal{H}$ of such properties $\mathcal{G}$ and $\mathcal{H}$ over a set of common variables. The main contribution is a particular proof rule based intuitively upon the concept of *coinduction*. This rule allows the inductive step of assuming that an entailment holds during the proof the entailment. In general, the proof method is based on an unfolding (and no folding) algorithm that reduces recursive definitions to a point where only constraint solving is necessary. The constraint-based proof obligation is then discharged with available solvers. The algorithm executes the proof by a search-based method which automatically discovers the opportunity of applying induction instead of the user having to specify some induction schema, and which does not require any base case.

1 Introduction

A large category of formal verification problems can be expressed as proof obligations of the form $\mathcal{G}$ *entails* $\mathcal{H}$, written $\mathcal{G} \models \mathcal{H}$, where $\mathcal{G}$ and $\mathcal{H}$ are recursively defined properties. Such problems appear in functional and logic programs, and specification languages such as JML, and they usually represent verification requirements for systems with infinite, or unbounded number of states, such as parameterized, or software systems. For instance, $\mathcal{G}$ might represent the semantics of a program, expressed as a formula in a suitable theory, whereas $\mathcal{H}$ may express a safety assertion.

Once the proof obligation $\mathcal{G} \models \mathcal{H}$ is formulated, it may be discharged with the help of a theorem prover such as Coq [1], HOL [6], or PVS [20]. While, in general, the proof process may be very complex, these tools provide a high level of assistance, automating parts of the process, and guaranteeing the correctness of the proof, once it is obtained. While there is, currently, a sustained research effort towards automating the process of discharging proof obligations, this process still requires, in general, a significant level of manual input. In the case of inductive proofs, for instance, the inductive variable, its base case, and the induction hypothesis need to be provided manually.

In this paper we present a proof method that establishes an entailment of the form $\mathcal{G} \models \mathcal{H}$, where $\mathcal{G}$ and $\mathcal{H}$ are two recursively defined properties over a set of common variables. The use of a coinduction principle (which does not require a base case), coupled with the standard operation of unfolding recursive definitions, allows the opportunistic discovery of suitable induction hypotheses, and makes our method amenable to automation. The entire framework is formalized in Constraint Logic Programming (CLP), so that CLP predicates can be used to describe the recursive properties of interest. Our method is, in fact, centered around an algorithm whose main operation is the standard unfolding of a CLP goal. The unfolding operation is applied to both the lhs $\mathcal{G}$ and the rhs $\mathcal{H}$ of the entailment. The principle of coinduction allows the discovery of a valid induction hypothesis, thus terminating the unfolding process. Through the application of the coinduction principle, the original proof obligation usually reduces to one that no longer contains recursive predicates. The remaining proof obligation contains only base constraints, and can be relegated to the underlying constraint solver.

Let us illustrate this process on a small example. Consider the definition of the following two recursive predicates

$$\begin{array}{ll} \mathit{m4}(0). & \mathit{even}(0). \\ \mathit{m4}(X+4) \texttt{ :- } \mathit{m4}(X). & \mathit{even}(X+2) \texttt{ :- } \mathit{even}(X). \end{array}$$

whose domain is the set of non-negative integers. The predicate *m4* defines the set of multiples of four, whereas the predicate *even* defines the set of even numbers. We shall attempt to prove that $\mathit{m4}(X) \models \mathit{even}(X)$, which in fact states that every multiple of four is even. We start the proof process by performing a *complete unfolding* on the lhs goal. By "complete," we mean that we use all the clauses whose head unify with $\mathit{m4}(X)$[1]. We note that $\mathit{m4}(X)$ has two possible unfoldings, one leading to the empty goal with the answer $X=0$, and another one leading to the goal $\mathit{m4}(X'), X'=X-4$. The two unfolding operations, applied to the original proof obligation result in the following two new proof obligations, both of which need to be discharged in order to prove the original one.

$$\begin{array}{rl} X=0 \models \mathit{even}(X) & (1) \\ \mathit{m4}(X'), X'=X-4 \models \mathit{even}(X) & (2) \end{array}$$

The proof obligation (1) can be easily discharged. Since unfolding on the lhs is no longer possible, we can only unfold on the rhs. We choose[1] to unfold with clause $\mathit{even}(0)$, which results in a new proof obligation which is trivially true, since its lhs and rhs are identical.

For proof obligation (2), before attempting any further unfolding, we note that the lhs $\mathit{m4}(X')$ of the current proof obligation, and the lhs $\mathit{m4}(X)$ of the original proof obligation, are unifiable (as long as we consider X' a fresh variable), which enables the application of the coinduction principle. First, we "discover" the *induction hypothesis* $\mathit{m4}(X') \models \mathit{even}(X')$, as a variant of the original proof obligation. Then, we use this induction hypothesis to replace $\mathit{m4}(X')$ in (2) by $\mathit{even}(X')$. This yields the new proof obligation

$$\mathit{even}(X'), X'=X-4 \models \mathit{even}(X) \quad (3)$$

To discharge (3), we unfold twice on the rhs, using the $\mathit{even}(X+2) \texttt{ :- } \mathit{even}(X)$ clause. The resulting proof obligation is

$$\mathit{even}(X'), X'=X-4 \models \mathit{even}(X'''), X'''=X''-2, X''=X-2 \quad (3)$$

where variables X'' and X''' are existentially quantified[2]. Using constraint simplification, we reduce this proof obligation to $\mathit{even}(X-4) \models \mathit{even}(X-4)$, which is obviously true.

[1] The requirement of a complete unfold on the lhs, and the lack of such requirement on the rhs, is explained in Section 3.

[2] In Section 3 we handle these variables formally.

At this point, all the proof obligations have been discharged and the proof is complete. Informally, we have performed four kinds of operations: (a) left unfolding, (b) right unfolding, (c) application of coinduction, and (d) constraint solving/simplification. While we shall relegate to Section 3 the argument that all these steps are correct, we would like to further emphasize several aspects concerning our proof method.

First, our method is amenable to automation, in the form of a non-deterministic algorithm. The state of the proof is given by a proof tree, whose frontier has the current proof obligations, all of which have to be discharged in order to complete the proof. Each proof step applies non-deterministically one of the four operations given above to one of the current proof obligations. Of these four, the lhs and rhs unfolding operations expand the tree by adding new descendants. In contrast, the coinduction operation searches the ancestors of the current goal for a matching lhs. If one is found, then a suitable induction hypothesis is generated, and applied to the lhs of the current goal, as shown in the small example given above. The fourth kind of operation performs constraint simplification/solving, possibly discharging the current proof obligation. As our examples show, the unfolding process and the application of the coinduction principle require no manual intervention.

Second, our coinductive proof step is inspired from tabled logic programming [24]. The intuition behind the correctness of this step is that, since the unfolding of the lhs is complete, we are already exploring all the possibilities of finding a counterexample, i.e. a substitution θ for which $\mathcal{G}\theta$ is true while $\mathcal{H}\theta$ is false. Whenever we find an ancestor with lhs $\mathcal{G}'$ which is variant of the lhs $\mathcal{G}$ (or some subgoal thereof) of the current proof obligation, we can immediately conclude that the current proof obligation would not contribute counterexamples that wouldn't already be visible from its matching ancestor. However, for this statement to be indeed true, we need to establish a similar matching between the rhs of the two proof obligations. This condition is expressed by the proof obligation obtained after the application of the coinductive step.

Finally, we would like to clarify that the use of the term *coinduction* pertains to the way the proof rules are employed for a proof obligation $\mathcal{G} \models \mathcal{H}$, and has no bearing on the *greatest* fixed point of the underlying logic program P. In fact, our proof method, when applied successfully, proves that $\mathcal{G}$ is a subset of $\mathcal{H}$ wrt. the *least* fixpoint of (the operator associated with) the program. However, as further clarified in Section 4, the success of the proof method is modeled as a property of a potentially infinite proof tree, and thus *coinduction*, rather than induction, needs to be employed to establish it.

1.1 Related Work

Variants of our proof method have been applied in more restricted settings of timed automata verification [10] and reasoning about structural properties of programs [12]. In the current paper, we focus on the common techniques used as well as hinting towards greater class of applications.

Among logic-programming-based proof methods, early works of [13, 14] propose *definite clause inference* and *negation as failure inference* (*NFI*) which are similar to our unfolding rules. These inferences are applied prior to concluding a proof of an implication using a form of computational induction. A form of structural induction in a similar framework is employed in [4]. We note that these proof methods are based on fitting in the allowable inductive proofs into an *induction schema*, which is usually syntax-based. Mesnard et al. [18] propose a CLP proof method for a system of implications, whose consequents contain only constraints. This technique is not completely general. Craciunescu [3] proposed a method to prove the equivalence of CLP programs using either induction or coinduction. The notion of coinduction here is different from ours; they reason about the *greatest* fixpoint of a CLP program, while we reason about the least.

Among the more automated approaches, [21, 22] used unfold/fold transformation of logic programs to prove equivalences of goals. [22] presents a proof method for equivalence assertions on parameterized systems. Hsiang and Srivas [7, 8] propose an inductive proof method for Prolog programs. The main feature of the proof method is a semi-automatic generation of induction schema (in the sense, this objective is similar to those of Kanamori and Fujita [13] mentioned above). The generation of inductive assertions is by producing the reduct of the goals (unfolding). Termination of the unfolding is implemented by a marking mechanism on the variables. Whenever an input variables is instantiated during an unfold (in other words, we need to make a decision about its value), it is marked. In a sense, this is similar to the use of *bomblist* in the Boyer-Moore prover [2]. As is the case with Boyer-Moore prover, the induction is structural. Here, the method requires the user to distinguish a set of *input* variables to structurally induct on. In comparison, we employ no induction schema. We detect the point where we apply the induction hypothesis automatically using constraint subsumption test. In other words, we discover the induction schema dynamically using indefinite steps of unfolds. This approach is more complete and automatable.

The work of Roychoudhury et al. [23] systematizes induction proofs using tabled resolution of logic programming. It is essentially based on unfolding, delaying upon detection of potential infinite resolution, and finally a folding step to conclude similarity. These serve to extend the tabled resolution engine of XSB tabled logic programming system. Our work generalizes this idea by providing a constraint-based inductive proof rules based on automated detection of cycles using constraints. Our rules are also based on the notion of tabling of assertions, which are later re-used as induction hypothesis.

Another form of tabling is also employed in *Prolog Technology Theorem Prover* (*PTTP*) [25]. Here the proof process is basically Prolog's search for refutation with several extensions, including a *model elimination reduction* (*ME reduction*), which memoes literals, and whenever a new goal which is contradictory to a stored literal is found, we stop because this constitutes a refutation. The part of PTTP that is akin to our coinduction is the detection when there is an occurrence of the same literal in which case, the system backtracks. Our work departs from PTTP mainly by the use of constraints.

Recursive definitions are also encountered in data structure verification area. [17] presents an algorithm for specification and verification of data structure using equality axioms. In [19] user-defined recursive definitions are allowed to specify "shape" properties. Proofs are carried out via fold/unfold transformations. As we will exemplify later, our algorithm can be used to automatically perform proofs of assertions containing recursive data structure definitions.

Finally, we mention the work in [5], which uses a coinductive interpretation of logic programming rules to express properties of infinite or circular data structures. The term *coinductive* is used here to refer to the greatest fixed point of the program at hand. We re-emphasize at this point that, in contrast with [5], our use of the term "coinductive" refers to the way our proof rules are employed, and bears no direct relationship to the greatest fixed point of the logic program.

2 Constraint Logic Programs

We use CLP [9] definitions to represent our verification conditions. To keep our paper self-contained, we provide a minimal background on the constraint logic programming framework.

An *atom* is of the form $p(\tilde{t})$ where p is a user-defined predicate symbol and $\tilde{t}$ a tuple of terms, written in the language of an underlying constraint solver. A *clause* is of the form $A \texttt{:-} \Psi, \tilde{B}$ where the atom A is the *head* of the clause, and the sequence of atoms $\tilde{B}$ and constraint Ψ constitute the *body* of the clause. The constraint Ψ is also written in the language of the underlying constraint solver, which is assumed to be able to decide (at least reasonably frequently) whether Ψ is satisfiable or not. In our examples, we assume an integer and array constraint solver, as described below.

A *program* is a finite set of clauses. A *goal* has exactly the same format as the body of a clause. A goal that contains only constraints and no atoms is called *final*.

A *substitution* θ simultaneously replaces each variable in a term or constraint e into some expression, and we write $e\theta$ to denote the result. A *renaming* is a substitution which maps each variable in the expression into a distinct variable. A *grounding* is a substitution which maps each integer or array variable into its intended universe of discourse: an integer or an array. Where Ψ is a constraint, a grounding of Ψ results in *true* or *false* in the usual way.

A *grounding* θ of an atom $p(\tilde{t})$ is an object of the form $p(\tilde{t}\theta)$ having no variables. A grounding of a goal $\mathcal{G} \equiv (p(\tilde{t}), \Psi)$ is a grounding θ of $p(\tilde{t})$ where $\Psi\theta$ is *true*. We write $[\![\mathcal{G}]\!]$ to denote the set of groundings of $\mathcal{G}$.

Let $\mathcal{G} \equiv (B_1, \cdots, B_n, \Psi)$ and P denote a non-final goal and program respectively. Let $R \equiv A \texttt{:-} \Psi_1, C_1, \cdots, C_m$ denote a clause in P, written so that none of its variables appear in $\mathcal{G}$. Let the equation $A = B$ be shorthand for the pairwise equation of the corresponding arguments of A and B. A *reduct* of $\mathcal{G}$ using a clause R, denoted *reduct*$(\mathcal{G}, R)$, is of the form

$(B_1, \cdots, B_{i-1}, C_1, \cdots, C_m, B_{i+1}, \cdots, B_n, B_i = A, \Psi, \Psi_1)$

provided the constraint $B_i = A \wedge \Psi \wedge \Psi_1$ is satisfiable.

A *derivation sequence* for a goal $\mathcal{G}_0$ is a possibly infinite sequence of goals $\mathcal{G}_0, \mathcal{G}_1, \cdots$ where $\mathcal{G}_i, i > 0$ is a reduct of $\mathcal{G}_{i-1}$. If the last goal $\mathcal{G}_n$ is a final goal, we say that the derivation is *successful*. A *derivation tree* for a goal is defined in the obvious way.

Definition 1 (Unfold). *Given a program P and a goal $\mathcal{G}$,* UNFOLD$(\mathcal{G})$ *is* $\{\mathcal{G}' | \exists R \in P : \mathcal{G}' = reduct(\mathcal{G}, R)\}$. $\square$

In the formal treatment below, we shall assume, without losing generality, that goals are written so that atoms contain only distinct variables as arguments.

2.1 An Integer and Array Constraint Language

In this section we provide a short description of constraint language allowed by the underlying constraint solver assumed in all our examples. We consider three kinds of terms: integer and array terms. Integer terms are constructed in the usual way, with one addition: the array element. The latter is defined recursively to be of the form $a[i]$ where a is an *array expression* and i an integer term. An array expression is either an array variable or of the form $\langle a, i, j \rangle$ where a is an array expression and i, j are integer terms. A term is either constructed from an array "segment": $a\{i..j\}$ where a is an array expression and i, j integer variables.

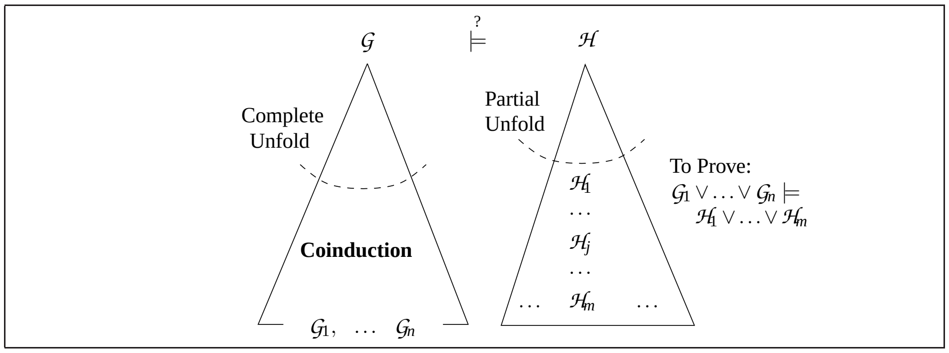

Fig. 1: Informal Structure of Proof Process

The meaning of an array expression is simply a map from integers into integers, and the meaning of an array expression $a' = \langle a, i, j \rangle$ is a map just like a except that $a'[i] = j$. The meaning of array elements is governed by the classic McCarthy [16] axioms:

$i = k \ \rightarrow \langle a, i, j \rangle[k] = j$
$i \neq k \ \rightarrow \langle a, i, j \rangle[k] = a[k]$

A *constraint* is either an integer equality or inequality, an equation between array expressions. The meaning of a constraint is defined in the obvious way.

In what follows, we use constraint to mean either an atomic constraint or a conjunction of constraints. We shall use the symbol ψ or Ψ, with or without subscripts, to denote a constraint.

3 Proof Method for Recursive Assertions

3.1 Overview

In this key section, we consider proof obligations of the form $\mathcal{G} \models \mathcal{H}$ where $var(\mathcal{H}) \subseteq var(\mathcal{G})$. The validity of this formula expresses the fact that $\mathcal{H}\theta$ succeeds w.r.t. the CLP program at hand whenever $\mathcal{G}\theta$ succeeds, for any grounding θ of $\mathcal{G}$. They are the central concept of our proof system, by being expressive enough to capture interesting properties of data structures, and yet amenable to automatic proof process.

Intuitively, we proceed as follows: unfold $\mathcal{G}$ completely a finite number of steps in order to obtain a "frontier" containing the goals $\mathcal{G}_1, \ldots, \mathcal{G}_n$. Then unfold $\mathcal{H}$, but this time not necessarily completely, that is, not necessarily obtaining *all* the reducts each time, obtain goals $\mathcal{H}_1, \ldots, \mathcal{H}_m$. This situation is depicted in Figure 1. Then, the proof holds if

$$\mathcal{G}_1 \vee \ldots \vee \mathcal{G}_n \ \models \ \mathcal{H}_1 \vee \ldots \vee \mathcal{H}_m$$

or alternatively, $\mathcal{G}_i \ \models \ \mathcal{H}_1 \vee \ldots \vee \mathcal{H}_m$ for all $1 \leq i \leq n$. This follows from the fact that $\mathcal{G} \models \mathcal{G}_1 \vee \ldots \vee \mathcal{G}_n$, (which is not true in general, but true in the least-model semantics of CLP), and the fact $\mathcal{H}_j \models \mathcal{H}$ for all j such that $1 \leq j \leq m$. More specifically, but with some loss of generality, the proof holds if

$$\forall i : 1 \leq i \leq n, \exists j : 1 \leq j \leq m : \mathcal{G}_i \ \models \ \mathcal{H}_j$$

and for this reason, our *proof obligation* shall be defined below to be simply a pair of goals, written $\mathcal{G}_i \ \models \ \mathcal{H}_j$.

3.2 The Proof Rules

We now present a formal calculus for the proof of $\mathcal{G} \models \mathcal{H}$. To handle the possibly infinite unfoldings of $\mathcal{G}$ and $\mathcal{H}$, we shall depend on the use of a key concept: *coinduction*. Proof by coinduction allows us to assume the truth of a *previous* obligation. The proof proceeds by manipulating a set of *proof obligations* until it finally becomes empty or a counterexample is found. Formally, a *proof obligation* is of the form $\tilde{A} \vdash \mathcal{G} \models \mathcal{H}$ where the $\mathcal{G}$ and $\mathcal{H}$ are goals and $\tilde{A}$ is a set of *assumption* goals. The role of proof obligations is to capture the state of a proof. The set $\tilde{A}$ contains goals whose truth can be assumed coinductively to discharge the proof obligation at hand. This set is implemented in our algorithm using a table as described in the next section.

Our proof rules are presented in Figure 2. The $\uplus$ symbol represents the disjoint union of two sets, and emphasizes the fact that in an expression of the form $A \uplus B$, we have that $A \cap B = \emptyset$. Each rule operates on the (possibly empty) set of proof obligations Π, by selecting one of its proof obligations and attempting to discharge it. In this process, new proof obligations may be produced.

The *left unfold with new induction hypothesis* (LU+I) (or simply "left unfold") rule performs a complete unfold on the lhs of a proof obligation, producing a new set of proof obligations. The original assertion, while removed from Π, is added as an assumption to every newly produced proof obligation, opening the door to using coinduction later in the proof.

The rule *right unfold* (RU) performs an unfold operation on the rhs of a proof obligation. In general, the two unfold rules will be systematically interleaved. The resulting proof obligations are then discharged either coinductively or directly, using the (CO) and (CP) rules, respectively.

The rule *coinduction application* (CO) transforms an obligation by using an assumption, and thus opens the door to discharging that obligation via the direct proof (CP) rule. Since assumptions can only be created using the (LU+I) rule, the (CO) rule realizes the coinduction principle. The underlying principle behind the (CO) rule is that a "similar" assertion $\mathcal{G}' \models \mathcal{H}'$ has been previously encountered in the proof process, and assumed as true.

Note that this test for coinduction applicability is itself of the form $\mathcal{G} \models \mathcal{H}$. However, the important point here is that this test can only be carried out using constraints, in the manner prescribed for the CP rule described below. In other words, this test does not use the definitions of assertion predicates.

Finally, the rule *constraint proof* (CP), when used repeatedly, discharges a proof obligation by reducing it to a form which contains no assertion predicates. Note that one application of this removes one occurrence of a predicate $p(\tilde{y})$ appearing in the rhs of an obligation. Once a proof obligation has no predicate in the rhs, a *direct proof* (DP) may be attempted by simply removing any predicates in the corresponding lhs.

Given a proof obligation $\mathcal{G} \models \mathcal{H}$, a proof shall start with $\Pi = \{\emptyset \vdash \mathcal{G} \models \mathcal{H}\}$, and proceed by repeatedly applying the rules in Figure 2 to it.

3.3 The Algorithm

We now describe a strategy so as to make the application of the rules automated. Here we propose systematic interleaving of the left-unfold (LU+I) and right-unfold (RU) rules, attempting a constraint proof along the way. As CLP can be execution by resolution, we can also execute our proof rules, based on an algorithm which has some resemblance to tabled resolution.

$$(\textsc{lu+i})\quad \frac{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}\}}{\Pi \,\cup\, \bigcup_{i=1}^{n} \{\tilde{A} \cup \{\mathcal{G} \models \mathcal{H}\} \vdash \mathcal{G}_i \models \mathcal{H}\}} \quad \begin{array}{l}\textsc{unfold}(\mathcal{G}) = \\ \{\mathcal{G}_1, \ldots, \mathcal{G}_n\}\end{array}$$

$$(\textsc{ru})\quad \frac{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}\}}{\Pi \cup \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}'\}} \quad \mathcal{H}' \in \textsc{unfold}(\mathcal{H})$$

$$(\textsc{co})\quad \frac{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}\}}{\Pi \cup \{\emptyset \vdash \mathcal{H}'\theta \models \mathcal{H}\}} \quad \begin{array}{l}\mathcal{G}' \models \mathcal{H}' \in \tilde{A} \text{ and there exists} \\ \text{a substitution } \theta \text{ s.t. } \mathcal{G} \models \mathcal{G}'\theta.\end{array}$$

$$(\textsc{cp})\quad \frac{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \wedge p(\tilde{x}) \models \mathcal{H} \wedge p(\tilde{y})\}}{\Pi \cup \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H} \wedge \tilde{x} = \tilde{y}\}} \qquad (\textsc{dp})\quad \frac{\Pi \uplus \{\mathcal{G} \models \mathcal{H}\}}{\Pi} \quad \begin{array}{l}\mathcal{G} \models \mathcal{H} \text{ holds by} \\ \text{constraint solving}\end{array}$$

Fig. 2: Proof Rules

We present our algorithm in pseudocode in Figure 3. Note that the presentation is in the form of a nondeterministic algorithm, and thus each of the nondeterministic operator **choose** needs to be implemented by some form of systematic search. Note also that when applying coinduction step, we test that some assertion $\mathcal{G}' \models \mathcal{H}'$ is stored in some table.

In Figure 3, by a *constraint proof* of a obligation, we mean to repeatedly apply the CP rule in order to remove all occurrences of assertion predicates in the obligation, in an obvious way. Then the constraint solver is applied to the resulting obligation.

3.4 Correctness

Given a proof obligation $\mathcal{G} \models \mathcal{H}$, a proof starts with $\Pi = \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}\}$, and proceeds by repeatedly applying the rules in Figure 2. The omission of negative literals in the body of the clauses of program P ensures that it has a unique *least model*, denoted $lm(P)$.

Theorem 1 (Soundness). *A proof obligation $\mathcal{G} \models \mathcal{H}$ holds, that is, $lm(P) \rightarrow (\mathcal{G} \models \mathcal{H})$ for the given program P, if, starting with the proof obligation $\emptyset \vdash \mathcal{G} \models \mathcal{H}$, there exists a sequence of applications of proof rules that results in proof obligations $\tilde{A} \vdash \mathcal{G}' \models \mathcal{H}'$ such that (a) $\mathcal{H}'$ contains only constraints, and (b) $\mathcal{G}' \models \mathcal{H}'$ can be discharged by the constraint solver.* $\square$

Proof Outline. The rule (RU) is sound because by the semantics of CLP, when $\mathcal{H}' \in \textsc{unfold}(\mathcal{H})$ then $\mathcal{H}' \models \mathcal{H}$. Therefore, the proof of the obligation $\tilde{A} \vdash \mathcal{G} \models \mathcal{H}$ can be replaced by the proof of the obligation $\tilde{A} \vdash \mathcal{G} \models \mathcal{H}'$ since $\mathcal{G} \models \mathcal{H}'$ is stronger than $\mathcal{G} \models \mathcal{H}$. Similarly, the rule (CP) is sound because $\mathcal{G} \models \mathcal{H} \wedge \tilde{x} = \tilde{y}$ is stronger than the $\mathcal{G} \wedge p(\tilde{x}) \models \mathcal{H} \wedge p(\tilde{y})$.

The rule (LU+I) is *partially sound* in the sense that when $\textsc{unfold}(\mathcal{G}) = \{\mathcal{G}_1, \ldots, \mathcal{G}_n\}$, then proving $\mathcal{G} \models \mathcal{H}$ can be substituted by proving $\mathcal{G}_1 \models \mathcal{H}, \ldots, \mathcal{G}_n \models \mathcal{H}$. This is because in the least-model semantics of CLP, $\mathcal{G}$ is equivalent to $\mathcal{G}_1 \vee \ldots \vee \mathcal{G}_n$. However, whether the addition of $\mathcal{G} \models \mathcal{H}$ to the set of assumed assertions $\tilde{A}$ is sound depends on the use of the set of assumed assertions in the application of (CO).

```
REDUCE($\mathcal{G} \models \mathcal{H}$) returns boolean

• Constraint Proof: (CP) + Constraint Solving (DP)
      Apply a constraint proof to $\mathcal{G} \models \mathcal{H}$.
      If successful, return true, otherwise return false
• Memoize $(\mathcal{G} \models \mathcal{H})$ as an assumption
• Coinduction: (CO)
      if there is an assumption $\mathcal{G}' \models \mathcal{H}'$ such that
      REDUCE($\mathcal{G} \models \mathcal{G}'\theta$) = true and REDUCE($\mathcal{H}'\theta \models \mathcal{H}$) = true then return true.
• Unfold:
      choose left or right
      case: Left: (LU+I)
            choose an atom $A$ in $\mathcal{G}$ to reduce
            for all reducts $\mathcal{G}_L$ of $\mathcal{G}$ using $A$:
                  if REDUCE($\mathcal{G}_L \models \mathcal{H}$) = false return false
            return true
      case: Right: (RU)
            choose an atom $A$ in $\mathcal{H}$ to reduce, obtaining $\mathcal{G}_R$
            return REDUCE($\mathcal{G} \models \mathcal{G}_R$)
```

Fig. 3: The Algorithm

Notice that in the rule (CO) we require the proofs of both $\mathcal{G} \models \mathcal{G}'\theta$ and $\mathcal{H}'\theta \models \mathcal{H}$ for some substitution θ. These proofs establish *subsumption*, that is the implication $(\mathcal{G}' \models \mathcal{H}') \rightarrow (\mathcal{G} \models \mathcal{H})$.

Assume that using our method, given a program P, we managed to conclude $\mathcal{G} \models \mathcal{H}$ where $\mathcal{G}$ and $\mathcal{H}$ are goals possibly containing atoms and it is not the case that $\mathcal{G} \models \mathcal{H}$ can be proved without the application of (LU+I) (since otherwise trivial by soundness of (RU) and (CP)). Assume that in the proof, there are a number of assumed assertions $A_1, \ldots, A_n$ used coinductively as induction hypotheses. This means that in the proof of $\mathcal{G} \models \mathcal{H}$ the left unfold rule (LU+I) has been applied at least once (possibly interleaved with the applications of (RU) and (CP)) obtaining two kinds of assertions:

1. Assertions C which are directly proved using (RU), (CP), and constraint solving (DP).
2. Assertions B which are proved using (CO) step using some assumed assertion A_j as hypothesis for $1 \leq j \leq n$.

We may conclude that $\mathcal{G} \models \mathcal{H}$ holds. We now outline the proof of this.

First, define a *refutation* to an assertion $\mathcal{G} \models \mathcal{H}$ as a successful derivation of one or more atoms in $\mathcal{G}$ whose answer Ψ has an instance (ground substitution) θ such that $\Psi\theta \wedge \mathcal{H}\theta$ is false. A finite refutation corresponds to a such derivation of finite length. A nonexistence of finite refutation means that $lm(P) \rightarrow (\mathcal{G} \models \mathcal{H})$. A derivation of an atom is obtainable by left unfold (LU+I)) rule only. Hence a finite refutation of length k implies a corresponding k left unfold (LU+I) applications that results in a contradiction.

Due to:

1. the soundness of other rules (RU) and (CP) and the partial soundness of (LU+I) with the fact that A_i for all $1 \leq i \leq n$ is obtained from $\mathcal{G} \models \mathcal{H}$ by applying these rules, and
2. all assertions C are proved by (RU), (CP) and constraint solving (DP) alone,

we have: $\mathcal{G} \models \mathcal{H}$ holds if A_i holds for all $1 \leq i \leq n$, and this holds iff for all i such that $1 \leq i \leq n$, and for all $k \geq 0 : A_i$ has no finite refutation of length k.

We prove inductively:

- **Base case:** When $k = 0$, for all i such that $1 \leq i \leq n$, A_i trivially has no finite refutation of length 0.
- **Inductive case:** Assume that for all i such that $1 \leq i \leq n$, A_i has no finite refutation of length k or less $(*)$, we want to prove that for all i such that $1 \leq i \leq n$, A_i has no finite refutation of length $k+1$ or less $(**)$.
  Notice again in our assumptions above that assertions B are proved by applying (CO) using A_j for some $1 \leq j \leq n$. Because subsumption holds in every application of (CO), this means that for such B, $A_j \rightarrow B$. $(***)$.
  The proof is by contradiction. Now suppose that $(**)$ is false, that is, A_i for some i such that $1 \leq i \leq n$ has a finite refutation of length $k+1$ or less. But due to our hypothesis $(*)$, A_i has no finite refutation of length k or less. Therefore it must be the case that A_i has a finite refutation of length $k+1$.
  Again, note that we have applied (LU) to A_i at least once on the resulting assertions, possibly interleaved with applications of (RU) and (CP) obtaining the following two kinds of assertions:
  1. Assertions C which are proved by applications of (RU) and (CP) and constraint solving alone.
  2. Assertions B which are proved by (CO) using some A_j for $1 \leq j \leq n$ in the set of assumed assertions as induction hypothesis.

  Then in the above set of assertions, either:
  1. Some assertion of type C is a refutation to A_i of length $k+1$. However, regardless of the length, since all such assertions C are already proved by (RU), (CP), and constraint solving, this case is not possible.
  2. Since A_i has to have a finite refutation of length $k+1$, therefore there has to be at least one assertion of type B that is reached in k or less unfolds. Therefore, B has to have a refutation of length k or less. Now since subsumption $(***)$ holds, then it should be the case that some A_j for $1 \leq j \leq n$ such that $A_j \rightarrow B$ also has a finite refutation of length k or less. But this contradicts our hypothesis $(*)$ that A_i for all $1 \leq i \leq n$ has no finite refutation of length k or less. □

We finally mention that the proof rules are not *complete*. For example, when we have a program

$$p(X) \text{ :- } 0 \leq X \leq 3.$$
$$q(X) \text{ :- } 0 \leq X \leq 2.$$
$$q(X) \text{ :- } 1 \leq X \leq 3.$$

obviously $p(X) \models q(X)$ holds, but we cannot prove this using our rules. The reason is that $0 \leq X \leq 3$ (obtained from the unfold of $p(X)$) does not imply either $0 \leq X \leq 2$ or $1 \leq X \leq 3$ (both obtained by right unfolding $q(X)$). It is possible, however, to introduce new rules toward achieving completeness. For proving the above assertion, we could introduce a splitting of an assertion. For instance, we may split $\mathcal{G} \models \mathcal{H}$ into $\mathcal{G}, \phi \models \mathcal{H}$ and $\mathcal{G}, \neg\phi \models \mathcal{H}$ (ϕ in our example would be, say, $X \leq 1$). However, this is beyond the scope of this paper.

4 On the Coinduction Rule

Consider again the predicate *even* presented in Section 1. We now demonstrate a simple application of our rules to prove a property on the predicate. Consider proving the

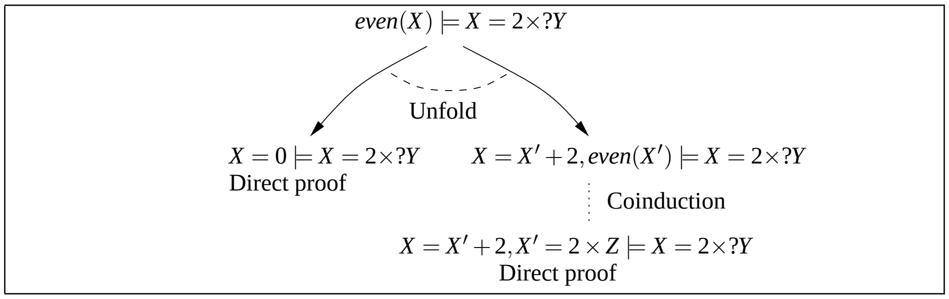
$even(X) \models X = 2\times?Y$
Unfold
$X = 0 \models X = 2\times?Y$
Direct proof
$X = X' + 2, even(X') \models X = 2\times?Y$
Coinduction
$X = X' + 2, X' = 2 \times Z \models X = 2\times?Y$
Direct proof

Fig. 4: Proof Tree Example

assertion $even(X) \models X = 2\times?Y$, call it A (we denote existentially-quantified variables with the query symbol "?"). The proof process starts by applying the (LU+I) rule unfolding the $even(X)$ goal, resulting in two new proof obligations, each with the original goal A as its assumption. On the left branch, after unfolding with the base-case clause, we are left with $X = 0 \models X = 2\times?Y$, which can be discharged by direct proof using a constraint solver. On the right branch of the proof, the unfolding rule produces the proof obligation $even(X'), X = X' + 2 \models X = 2\times?Y$. Here we apply the coinduction (CO) rule using $even(X) \models X = 2\times?Y$ as induction hypothesis, spawning an obligation to prove $X' = 2 \times Z, X = X' + 2 \models X = 2\times?Y$. This can then be proved using constraint solving.

Let us now recall our example in Section 1. In Section 1 we have applied (LU+I) to unfold the predicate $m4(X)$ resulting in the two obligations (1) and (2). We apply (RU) to perform right unfold on (1). We apply (CO) to (2) obtaining (3). We then apply (RU) to (3) twice to establish it.

Our system does not require the user to manually specify induction hypothesis and/or construct induction schema. Instead, any induction hypothesis used is obtained dynamically during the proof process. Let us now exemplify this concept by considering the program

```
p(X) :- q(X).
q(X) :- q(X).
r(X).
```

Here we want to prove $p(X) \models r(X)$. Call this A_1. We first apply (LU+I) to the assertion obtaining $q(X) \models r(X)$. Call this assertion A_2. At this point, our algorithm tests whether A_1 can be used as a induction hypothesis to establish A_2. This fails, and we again apply (LU+I) obtaining another assertion A_3 which is equivalent to A_2. Upon obtaining A_3, the set of assumed assertions contain both A_1 and A_2. The algorithm now tests whether any of these can be used in a (CO) application. Indeed, we can use the assertion A_2 which is identical to A_3. In this way induction hypotheses are chosen dynamically.

In the preceding examples we have demonstrated the use of the rule (CO) to conclude proofs. Moreover, the last example illustrates the fact that, in contrast to most inductive proof methods, our proof process may be successful even in the absence of a base case. While the lack of a base case requirement justifies the qualifier "coinductive" being applied to our proof method, the fact that this term has been somewhat overused in the logic programming community warrants further clarification.

In our view, induction and coinduction are two flavours of one general proof scheme, which is used to prove properties of objects defined by means of recursive rules. This general scheme proves properties of such objects by assuming that the property of inter-

Program:	**CLP Model:**
$F(x) \Leftarrow$ **if** $p(x)$ **then** x	$s(X,X) :\!- X = error.$
else $F(F(h(x)))$	$s(X,X_f) :\!- X \neq error, p(X) = 1, X_f = X.$
	$s(X,X_f) :\!- X \neq error, p(X) = 0, s(h(X),Y), s(Y,X_f).$

Fig. 5: Idempotent Function

est already (inductively) holds for the "smaller" object on which the definition recurses. Now, recursive definitions may be interpreted in an inductive or coinductive manner, and each of these interpretations would lead to the general proof scheme being construed as either induction or coinduction.

The crux of our proof method is to automatically generate an induction hypothesis for the goal at hand, in an attempt to produce a successful application of the general proof scheme mentioned above. The method works correctly irrespective of whether the rules defining the properties of interest are intepreted inductively or coinductively[3]. Since our proof method does not explicitly look for base cases, and since it can also handle the situation where a recursive definition of a property would be interpreted coinductively, we have chosen to use the qualifier "coinductive." However, this qualifier bears no direct relationship to the greatest fixed point of the logic program at hand. Throughout this paper, our recursive definitions are meant to be interpreted inductively, and the meaning of the goal $\mathcal{G} \models \mathcal{H}$ is that whenever a grounding $\mathcal{G}\theta$ lies in the *least* fixed point of the program at hand P, it follows that the grounding $\mathcal{H}\theta$ is also in $lfp(P)$. Our proof method will be successful only when this interpretation of a goal holds.

5 Proof Examples

In our driving examples area of program verification, most of the entailment problems we have encountered can be proved by our algorithm automatically. We believe they cannot be automatically discharged by any existing systematic method. In this section, we present two examples.

5.1 Function Idempotence

Suppose that we have the function in Figure 5 [15] with its CLP representation. Note that *error* represents the return value of the function on divergent termination. Here we want to prove idempotence, that is $F(x) = F(F(x))$, or that both the assertions A) $s(X,Y), s(Y,X_f) \models s(X,X_f)$ and B) $s(X,X_f) \models s(X,?Y), s(?Y,X_f)$ holds. The mechanical proof of Assertion A requires coinduction and will be exemplified here. The algorithm first applies (LU+I) obtaining the assertions 1) $s(error,X_f) \models s(error,X_f)$, 2) $X \neq error, p(X) = 1, X = Y, s(Y,X_f) \models s(X,X_f)$, and 3) $X \neq error$, $p(X) = 0$, $s(h(X),Z)$, $s(Z,Y)$, $s(Y,X_f) \models s(X,X_f)$. Assertions 1 and 2 are proved by (CP) and (DP), and the algorithm attempts to apply (CO) to Assertion 3 using the ancestor Assertion A as hypothesis.

The application of (CO) obtains the obligation $X \neq error$, $p(X) = 0$, $s(h(X),Y)$, $s(Y,X_f) \models s(X,X_f)$. This assertion cannot be proved by constraint proof nor by coinduction (since the set of assumed assertions are empty), and the algorithm proceeds

[3] Nevertheless, the complete unfold of the left goal ensures that correct base case proofs are generated whenever the current recursive definition provides such base cases.

to proving by unfolding. Here it applies right unfold (RU) rule obtaining $X \neq error$, $p(X) = 0$, $s(h(X),Y)$, $s(Y,X_f) \models X \neq error$, $p(X) = 0$, $s(h(X),?Z)$, $s(?Z,X_f)$, which can be proved directly. Since the application of (CO) to Assertion 3 has been successful, the proof concludes.

5.2 A Pointer Data Structure Example: List Reset

We represent pointers as indices in an array which we call the *heap*. We write $[p]$ to refer to the location referenced by the pointer p. To implement a linked list, we shall assume that a list element is made up of two adjacent heap cells. Thus, for the list element referenced by p, the data field is $[p]$, and the reference to the next element is $[p+1]$. In the CLP program, given an array H, which typically denotes the heap, we denote by $H[I]$ the element referenced by index I in the array. We also denote by $\langle H,I,J\rangle$ the array that is identical to H for all indices, except I, where the original value is replaced by J. The steps for solving constraints containing these constructs are discussed in [11].

Figure 6 shows a program which "zeroes" all elements of a given linked list with head `p`. We prove that the program produces a nonempty null-terminating list with zero values. Note that in Figure 6, `h` is a program variable denoting the current heap. The predicate takes into consideration the memory model of the program and expresses the relationship between the heap H before the execution of the program, and the heap H' obtained after the program has completed. Thus, the predicate $allz(H,H',L,R)$ states that the heap H' differs from H only by having zero elements in the non-empty sublist from L to R.

In Figure 6 we provide a *tail-recursive* definition of *allz* which defines a zeroed list segment (L,R) as one whose head contains zero, and its tail is, recursively, the zeroed list segment $(H[L+1],R)$[4]. We could have used a *sublist-recursive* specification: a zeroed list segment (L,R) is defined to be a zeroed list segment (L,T) appended by one extra zero element R. Clearly the program behaves in consistency with the latter definition, and not the former. We show that despite this, our method automatically discharges the proof.

Here we want to prove that $\Psi \equiv allz(h_0,\mathtt{h},p_0,\mathtt{p})$ is a loop invariant. Formally,

$$allz(H_0,H,P_0,P),H[P+1] > 0 \models allz(H_0,\langle H,H[P+1],0\rangle,P_0,H[P+1]). \qquad (Z.1)$$

For this assertion, constraint proof fails and coinduction (CO) is not applicable due to an empty set of assumed assertions. The algorithm applies left unfold (LU+I) using the definition of *allz* obtaining two new obligations, of which one is:

$$\begin{array}{l} allz(H_0,H_1,H_0[P_0+1],P),P_0 > 0,H_1[P+1] > 0 \models \\ \quad allz(H_0,\langle\langle H_1,P_0,0\rangle,H_1[P+1],0\rangle,P_0,H_1[P+1]). \end{array} \qquad (Z.2)$$

Now the algorithm applies (CO) using $Z.1$ as the hypothesis. As required by (CO), the algorithm spawns two sub-obligations, one of which proves

$$allz(H_0,H_1,H_0[P_0+1],P),P_0 > 0,H_1[P+1] > 0 \models allz(H_0,H_1,H_0[P_0+1],P),H_1[P+1] > 0$$

This is established by eliminating the predicates using (CP) and applying constraint solving to the following assertion:

$$P_0 > 0,H_1[P+1] > 0 \models H_0 = H_0,H_1 = H_1,H_0[P_0+1] = H_0[P_0+1],P = P,H_1[P+1] > 0.$$

The second sub-obligation is

$$\begin{array}{l} allz(H_0,\langle H_1,H_1[P+1],0\rangle,H_0[P_0+1],H_1[P+1]) \models \\ \quad allz(H_0,\langle\langle H_1,P_0,0\rangle,H_1[P+1],0\rangle,P_0,H_1[P+1]). \end{array} \qquad (Z.3)$$

[4] Note that we do not require that the list is acyclic ($L \neq R$).

Program:

```
    {h = h₀, p = p₀ > 0}
⟨0⟩ while (p>0) do
        [p] := 0 ⟨1⟩
        p := [p+1] ⟨2⟩ end ⟨3⟩
    {∃y.allz(h₀, h, p₀, y), h[y+1] = 0}
```

Assertion Predicate:

$allz(H, \langle H, L, 0\rangle, L, L)$:- $L > 0$.
$allz(H_1, \langle H_2, L, 0\rangle, L, R)$:-
  $L > 0, allz(H_1, H_2, H_1[L+1], R)$.

Fig. 6: List Reset

Here again the application of constraint proof and coinduction fails, and the algorithm performs a right unfold using the second clause of *allz* resulting in

$$\begin{aligned}&allz(H_0, \langle H_1, H_1[P+1], 0\rangle, H_0[P_0+1], H_1[P+1]) \models \\ &\quad allz(H_0, ?H_2, H_0[P_0+1], H_1[P+1]), \langle\langle H_1, P_0, 0\rangle, H_1[P+1], 0\rangle = \langle ?H_2, P_0, 0\rangle\end{aligned} \tag{Z.4}$$

By an application of (CP) proof rule, the algorithm removes the predicates and then solves the following implication by constraint solving (DP):

$$\begin{aligned}&true \models H_0 = H_0, H_0[P_0+1] = H_0[P_0+1], \\ &\quad H_1[P+1] = H_1[P+1], \langle\langle H_1, P_0, 0\rangle, H_1[P+1], 0\rangle = \langle\langle H_1, H_1[P+1], 0\rangle, P_0, 0\rangle.\end{aligned} \tag{Z.5}$$

6 Conclusion

We presented an automatic proof method which is based on unfolding recursive CLP definitions of user-specified program properties. The novel aspect is a principle of coinduction which is used in conjunction with a set of unfold rules in order to efficiently dispense recursive definitions into constraints involving integers and arrays. This principle is applied opportunistically and automatically over a dynamically generated set of potential induction hypotheses. As a result, we can now automatically discharge many useful proof obligations which previously could not be discharged without manual intervention. We finally demonstrated our method, assuming the use of a straightforward constraint solver over integers and integer arrays, to automatically prove two illustrative examples.

References

1. B. Barras, S. Boutin, C. Cornes, J. Courant, J. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. M. Noz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq proof assistant reference manual—version v6.1. Technical Report 0203, INRIA, 1997.
2. R. S. Boyer and J. S. Moore. Proving theorems about LISP functions. *J. ACM*, 22(1):129–144, 1975.
3. S. Craciunescu. Proving equivalence of CLP programs. In *18th ICLP*, volume 2401 of *LNCS*. Springer, 2002.
4. L. Fribourg. Automatic generation of simplification lemmas for inductive proofs. In *ISLP 1991*, pages 103–116. MIT Press, 1991.
5. G. Gupta, A. Bansal, R. Min, L. Simon, and A. Mallya. Coinductive logic programming and its applications. In *23rd ICLP*, volume 4670 of *LNCS*, pages 27–44. Springer, 2007.
6. J. Harrison. HOL light: A tutorial introduction. In M. K. Srivas and A. J. Camilleri, editors, *1st FMCAD*, volume 1166 of *LNCS*, pages 265–269. Springer, 1996.
7. J. Hsiang and M. Srivas. Automatic inductive theorem proving using Prolog. *TCS*, 54(1):3–28, 1987.

8. J. Hsiang and M. K. Srivas. A PROLOG framework for developing and reasoning about data types. In *1st TAPSOFT Vol. 2*, volume 186 of *LNCS*, pages 276–293. Springer, 1985.
9. J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *J. LP*, 19/20:503–581, May/July 1994.
10. J. Jaffar, A. Santosa, and R. Voicu. A CLP proof method for timed automata. In *25th RTSS*, pages 175–186. IEEE Computer Society Press, 2004.
11. J. Jaffar, A. E. Santosa, and R. Voicu. Recursive assertions for data structures. Available from `http://www.comp.nus.edu.sg/˜joxan/papers/rads.pdf`.
12. J. Jaffar, A. E. Santosa, and R. Voicu. Relative safety. In E. A. Emerson and K. S. Namjoshi, editors, *7th VMCAI*, volume 3855 of *LNCS*, pages 282–297. Springer, 2006.
13. T. Kanamori and H. Fujita. Formulation of induction formulas in verification of Prolog programs. In *8th CADE*, volume 230 of *LNCS*, pages 281–299. Springer, 1986.
14. T. Kanamori and H. Seki. Verification of Prolog programs using an extension of execution. In E. Y. Shapiro, editor, *3rd ICLP*, volume 225 of *LNCS*, pages 475–489. Springer, 1986.
15. Z. Manna, S. Ness, and J. Vuillemin. Inductive methods for proving properties of programs. *Comm. ACM*, 16(8):491–502, August 1973.
16. J. McCarthy. Towards a mathematical science of computation. In C. M. Popplewell, editor, *IFIP Congress 1962*. North-Holland, 1983.
17. S. McPeak and G. C. Necula. Data structure specifications via local equality axioms. In *17th CAV*, volume 3576 of *LNCS*, pages 476–490. Springer, 2005.
18. F. Mesnard, S. Hoarau, and A. Maillard. CLP($\mathcal{X}$) for automatically proving program properties. In F. Baader and K. U. Schulz, editors, *1st FroCoS*, volume 3 of *Applied Logic Series*. Kluwer Academic Publishers, 1996.
19. H. H. Nguyen, C. David, S. C. Qin, and W. N. Chin. Automated verification of shape and size properties via separation logic. In B. Cook and A. Podelski, editors, *8th VMCAI*, volume 4349 of *LNCS*. Springer, 2007.
20. S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *8th CAV*, volume 1102 of *LNCS*, pages 411–414. Springer, 1996.
21. A. Pettorossi and M. Proietti. Synthesis and transformation of logic programs using unfold/fold proofs. *J. LP*, 41(2–3):197–230, 1999.
22. A. Roychoudhury, K. N. Kumar, C. R. Ramakrishnan, and I. V. Ramakrishnan. An unfold/fold transformation framework for definite logic programs. *ACM TOPLAS*, 26(3):464–509, 2004.
23. A. Roychoudhury, C. R. Ramakrishnan, I. V. Ramakrishnan, and S. A. Smolka. Tabulation-based induction proofs with application to automated verification. In *1st TAPD*, pages 83–88, April 1998. URL http://pauillac.inria.fr/˜ clerger/tapd.html.
24. K. Sagonas, T. Swift, D. S. Warren, J. Freire, P. Rao, B. Cui, E. Johnson, L. de Castro, S. Dawson, and M. Kifer. *The XSB System Version 2.5 Volume 1: Programmer's Manual*, June 2003.
25. M. E. Stickel. A Prolog technology theorem prover: A new exposition and implementation in prolog. *TCS*, 104(1):109–128, 1992.