RACSAM
Rev. R. Acad. Cien. Serie A. Mat.
VOL. **98 (**1), 2004, pp. 153–180
Ciencias de la Computación / Computational Sciences

Induction and Decision Procedures

Deepak Kapur, Jürgen Giesl and Mahadevan Subramaniam

Abstract. Mechanization of inductive reasoning is an exciting research area in artificial intelligence and automated reasoning with many challenges. An overview of our work on mechanizing inductive reasoning based on the cover set method for generating induction schemes from terminating recursive function definitions and using decision procedures is presented. This paper particularly focuses on the recent work on integrating induction into decision procedures without compromising their automation.

Inducción y procedimientos de decisión

Resumen. Dentro del campo de la inteligencia artificial y del razonamiento automático, la mecanización del razonamiento inductivo es un área de investigación apasionante que se enfrenta a muchos retos. Se presenta una visión global del trabajo de los autores sobre la mecanización del razonamiento inductivo basado en el método de recubrimiento de conjuntos para la generación de esquemas de inducción a partir de definiciones de funciones recursivas con terminación que usa procedimientos de decisión. Este artículo se centra sobre todo en el trabajo reciente que se ha realizado con respecto a la integración de la inducción en los procedimientos de decisión sin comprometer su automatización.

1. Introduction

Over the past 30 years or so, there has been a great deal of interest in developing heuristics and techniques for mechanizing reasoning by induction, partly because proofs by induction must be carried out in many application areas including software and hardware verification, specification analysis, common-sense reasoning, and other aspects of artificial intelligence and mathematics. Many induction theorem provers have been developed including NQTHM [4], RRL [24], CLAM [7], ACL2 [6], INKA [1], and SPIKE [3]. Despite significant progress made in research on mechanization of inductive reasoning, using induction theorem provers for proofs of properties of recursive and iterative programs expressed in a functional style often requires significant manual intervention. A user can spend considerable time on proof attempts which fail due to the inherent incompleteness in inference systems for inductive reasoning (provers fail even for many valid conjectures). This is especially daunting to an application expert trying to use an induction prover for obvious conjectures.

There has recently been a surge of interest in the role of decision procedures in tools for reasoning about computations, especially because of the success of BDD-based tools and model checkers in hardware and protocol verification, static and type analyzes, byte-code verification, and proof-carrying codes. Most such push-button systems use a combination of decision procedures for theories such as Presburger arithmetic, propositional satisfiability, and data structures including bit vectors, arrays, and lists. Extending these

Presentado por Luis M. Laita.
Recibido: January 26, 2004. *Aceptado*: October 13, 2004.
Palabras clave / Keywords: Induction, Decision Procedures, Automated Theorem Proving.
Mathematics Subject Classifications: 3B35, 68Q42, 68Q60, 68T15.

tools by the capability to perform induction proofs would be very desirable, since induction is frequently needed to reason about structured and parameterized circuits (e.g., n-bit adders or multipliers), the timing behavior of circuits with feedback loops, and code using loops and/or recursion over recursively defined data structures. Because of the above-mentioned challenges in automating induction proofs, such tools lack, however, support for inductive reasoning.

This paper attempts to provide an overview of our work on mechanizing inductive reasoning using term rewriting techniques in the framework of our theorem prover *Rewrite Rule Laboratory* (*RRL*). A particular focus is our recent work on integrating induction into decision procedures so as to enhance the reasoning power of decision procedures. Below, we start with a brief history.

In his Ph.D. thesis in 1988, Kapur's former student Hantao Zhang proposed a method for mechanizing induction based on the concept of a *cover set* built from a definition of a function given using terminating rewrite rules [29]. This method was implemented in *RRL* [24]; it turned out to be quite successful and effective in proving nontrivial theorems about lists and numbers. Particularly, we managed to prove most of the theorems proved in [4], some of them automatically, and almost all of them with fewer interventions and guidance by the user than reported in [4]. Subsequently, Xumin Nie implemented a decision procedure for quantifier-free Presburger arithmetic into *RRL* and integrated it with the contextual rewriting mechanisms of *RRL* [14, 16]. This extension turned out to be particularly useful in automatic verification of number theoretic properties of generic parameterized descriptions of arithmetic hardware circuits including ripple-carry adder, carry-save adder, carry-lookahead adder, Wallace tree and related multipliers, and SRT division (see [18, 20, 21] for details). Many of the proofs could be done completely automatically using *RRL*. In particular, the theorem prover was able to generate intermediate lemmas needed for proofs.

In the early 1990's, Kapur and Subramaniam initiated a research program in mechanization of inductive reasoning in which decision procedures are to play a critical role [17]. The main idea in this work was to use decision procedures instead of syntactic (simple) unification over the empty theory, to generate cover sets, induction schemes, merge induction schemes, and speculate generalization lemmas. In the late 1990s, Kapur and Subramaniam proposed a way to extend decision procedures by integrating induction schemes, so as to be able to decide a priori the class of formulas where inductive validity can be verified or disproved. In this way, the language of these decision procedures is extended by recursively defined function symbols. Using these ideas, Giesl, Kapur, and Subramaniam identified a wide class of conjectures which can be decided automatically using the cover set method [11, 12, 22]. This partly explains why *RRL* has been so successful in automatically deciding many conjectures. We are however still a long way from characterizing the class of conjectures automatically decidable by *RRL*.

In this paper, we introduce key concepts and discuss results obtained; an interested reader should consult [11, 12, 17, 22, 28] for further details.

1.1. Organization

In the next subsection, an informal overview of the cover set method is presented using an example that students in an introductory course on discrete mathematics are typically asked to do when they are taught proofs by mathematical induction. This is followed by another example where the use of semantic information (with the help of a decision procedure for numbers) is illustrated. Section 2 gives preliminary background: how function definitions are given, axiomatization of recursive data structures including numbers is reviewed, inductive validity is defined. Section 3 defines the concept of a cover set; it is shown how induction schemes are generated from a cover set both in the most general case as well as in specific cases; soundness and completeness of an induction scheme are reviewed and a method for checking the completeness of a cover set is presented. It is also shown how incomplete induction schemes can be useful in doing proofs. This is followed by a discussion of merging induction schemes, which often becomes necessary while attempting proofs by induction.

The second part of the paper is on extending decision procedures with induction schemes. In Section 4, a class of equational conjectures in which defined function symbols occur is identified so that their validity can be decided using the cover set method and decision procedures. Section 5 is on identifying a class of

quantifier-free formulas which can be decided automatically. Their atomic formulas are a subclass of the above equational conjectures expressed only using defined symbols on one side and constructor terms on the other side. Section 6 discusses how a decision procedure can be used to automatically generate lemmas and to improve upon generalization heuristics.

1.2. An Informal Overview of Issues in Mechanizing Induction

Assume that functions $+, *$, and exponentiation are defined on the data type of natural numbers, generated by its free constructors $0, \mathsf{s}$ as follows:

$$\begin{array}{rclrcl} x+0 & \rightarrow & x, & x+\mathsf{s}(y) & \rightarrow & \mathsf{s}(x+y), \\ x*0 & \rightarrow & 0, & x*\mathsf{s}(y) & \rightarrow & (x*y)+x, \\ x^0 & \rightarrow & \mathsf{s}(0), & x^{\mathsf{s}(y)} & \rightarrow & x^y * x. \end{array}$$

¿From these definitions which can be easily shown to be terminating when considered as rewrite rules from left to right, we wish to prove:

$$x^{y+z} = x^y * x^z.$$

It is easy to see that the above conjecture cannot be established by simplification using the rules. Let us see what is involved in attempting its proof using induction; the example will reveal many issues in mechanizing proofs by induction.

The first issue is the choice of a variable(s) to perform induction; there are three candidates: x, y, z. A related issue is that of a possible induction scheme on natural numbers which must be used. By analyzing the definitions of function symbols in the conjecture, it becomes clear that a proof attempt is less likely to get stuck if (i) the variables on which recursion is being performed are used as the induction variables and (ii) further, an induction scheme based on recursion analysis of function definitions is used. Particularly, resulting subgoals in a proof attempt can be simplified using the definitions, possibly leading to formulas on which induction hypotheses can be used.

Based on such definitional analysis, x is ruled out as a possible candidate for performing induction. Among the remaining variables y and z, which are appearing as inductive arguments of exponentiation, z appears to be the most promising since it is the inductive argument on both side of the conjecture.

With respect to the choice of an induction scheme to be used for z, the definitions of exponentiation and $+$ in which z is the inductive argument are given in the same way.[1] Thus, the induction scheme suggested by $+$, which is the principle of mathematical induction, can be used.

We attempt a proof by induction using z as the induction variable. The base case, in which z is 0, gives the following subgoal: $x^{y+0} = x^y * x^0$, which simplifies using the definitions to: $x^y = 0 + x^y$. The proof of this new lemma can be attempted again by induction. Or, one can be speculative, suggesting that perhaps a more general lemma obtained by generalizing x^y to u may be valid. If that is indeed the case, then the less general subgoal obviously follows, and the more general lemma can be useful in other proof attempts as well (otherwise, it is always possible to attempt the less general goal).

Let us try to attempt a proof of $u = 0 + u$. In this case, there is no choice; u is the only variable. The induction scheme suggested by the definition of $+$ is employed, and the proof (both the base case and the induction step case) goes through easily. This also finishes the proof of the base case of the main conjecture.

In the induction step case, the conclusion after substituting $\mathsf{s}(z_1)$ for z is:

$$x^{y+s(z_1)} = x^y * x^{s(z_1)},$$

with $x^{y+z_1} = x^y * x^{z_1}$ as the induction hypothesis. The conclusion can be simplified using the definitions to: $x^{y+z_1} * x = x^y * (x^{z_1} * x)$. Now the induction hypothesis is applicable to the left-hand side of the above subgoal, giving rise to: $(x^y * x^{z_1}) * x = x^y * (x^{z_1} * x)$. Once again, it is possible to speculate and observe

[1] If that was not the case, it would become necessary to reconcile different induction schemes suggested by different definitions possibly by merging them; this is discussed later in the paper.

that perhaps a more general version of the formula can be attempted from which the formula follows. The generalized conjecture is:

$$(u * v) * x = u * (v * x),$$

which is a property about $*$. Again, this property cannot be proved by simplification, but induction must be used. Based on definitional analysis, v, x are candidates for induction, and since x appears on both sides of the conjecture as the inductive argument of $*$, it is picked for performing induction using the induction scheme suggested by $*$.

The base case, in which x becomes 0, follows by applying the definition. The induction step case gives rise to: $(u * v) * s(x_1) = u * (v * s(x_1))$, with the hypothesis: $(u * v) * x_1 = u * (v * x_1)$. Simplifying the induction subgoal using the definition and applying the induction hypothesis results in: $(u * (v * x_1)) + (u * v) = u * ((v * x_1) + v)$. Once again, a generalization of this conjecture can be attempted, which is the distributivity of $*$ over $+$.

$$(u * w) + (u * v) = u * (w + v).$$

Based on the analysis discussed above, its proof is attempted by induction using v as the induction variable with the induction scheme suggested by $*$ and $+$, the principle of mathematical induction. The base case is easy to prove. In the induction step case, the subgoal is: $(u*w)+(u*s(v_1)) = u*(w+s(v_1))$, with the induction hypothesis: $(u * w) + (u * v_1) = u * (w + v_1)$. Attempting a proof of the subgoal leads, after yet another generalization, to a conjecture about the associativity of $+$, which can be easily established using induction.

Once the associativity of $+$ is proved, the proof of the distributivity conjecture is complete, which also completes the proof of the associativity of $*$. This then completes the proof of the original conjecture. (This is exactly how the automatic proof generated by *RRL* proceeds.)

The above proof is a typical induction proof attempt which succeeded. It reveals a number of issues that need to be considered when proofs by induction are mechanized. We briefly review some of them below:

1. Which variable in a conjecture should be selected for performing induction? Associated with this choice is determining an induction scheme to be used. In this example, the choice of an induction scheme was not difficult, but there are cases where such a choice is not easy. Designing an appropriate induction scheme for attempting a proof is perhaps one of the most challenging tasks in automating induction proofs.

2. What techniques can be used for generalizing intermediate subgoals to identify stronger lemmas which are likely to be more useful and easier to prove? Further, if simple heuristics, such as generalizing a commonly occurring subterm to a variable, are employed, can conditions be identified under which the generalization is likely to be valid if the original conjecture is valid? Ideally, we would like to perform a *safe* generalization such that if the generalized conjecture is found to be not valid, the original conjecture can be declared to be not valid also.

3. When can a proof by induction be done automatically without any help by the user?

4. How to ensure progress during a proof attempt when subgoals are generated from the original goal while following a particular line of reasoning, and when should an alternate approach be attempted instead? Related to this issue is the number of generalizations attempted and the number of intermediate lemmas tried before giving up a particular line of reasoning.

5. When should one give up? How can the information gathered in a proof attempt be utilized for subsequent proof attempts of the conjecture or of the modified and related conjecture if the conjecture is not valid? How can a conjecture be patched?

The paper will attempt to address some of these issues focusing on 3.

1.3. Using Semantic Information in Attempting Proofs by Induction

Consider the following definition of the divisibility predicate on natural numbers; the predicate $<$ on numbers has the usual meaning and its definition is omitted.

$$\begin{array}{rcl} \mathsf{divides}(u, 0) & \rightarrow & \mathsf{true} \\ \mathsf{divides}(u, u+v) & \rightarrow & \mathsf{divides}(u, v) \text{ if } v < u+v. \end{array} \qquad \mathsf{divides}(u, v) \rightarrow \mathsf{false} \text{ if } v < u \wedge v \neq 0,$$

The above definition is not given using 0 and s. It illustrates how semantic information about a data structure is used for attempting proofs. A possible conjecture to prove is:

$$\mathsf{divides}(x, y+y) = \mathsf{true} \text{ if } \mathsf{divides}(x, y) \wedge x > 0.$$

The reader would notice that attempts to prove the above conjecture by induction on x or y or both x, y using the principle of structural mathematical induction on natural numbers will get stuck. The above definition of divides is not given using 0 and s. However, for such a proof attempt, it would become necessary to prove properties of divides expressed using 0 and s.

The definition of divides suggests an induction scheme different from the principle of structural mathematical induction. A few things must be observed first, however. Unlike the definitions in the previous subsection, for which it was easy to prove that each function is completely defined and terminating (since each definition follows the primitive recursive scheme), the above definition is nontrivial. As will be shown later, a formula can be constructed from the above definition which specifies its coverage:

$$\forall x, y\, \exists u, v\, [(x = u \wedge y = 0) \vee (x = u \wedge y = v \wedge v < u \wedge v \neq 0) \vee$$
$$(x = u \wedge y = u + v \wedge v < u + v)].$$

This formula is expressed using function and predicate symbols from Presburger arithmetic and can be decided by a decision procedure for this theory. The formula is not valid as there are values of x, y, particularly when $x = 0, y = \mathsf{s}(y_1)$ for which the above rules do not define divides; it is defined only if the first argument is non-zero or both the arguments are 0. Thus, any conjecture involving divides must satisfy this condition on the arguments. Since divides is not completely defined, the induction scheme suggested by the above definition can be used only to prove conjectures under a condition as shown below.

Since in the definition of divides, both the arguments are changing, variables x and y are the induction variables in the above conjecture. There are two base cases corresponding to the first two rewrite rules: (i) $x = u$ and $y = 0$, and (ii) $x = u$ and $y = v$ where $(v < u \ \wedge \ v \neq 0)$. The induction step case is generated from the third rewrite rule in which a recursive call is made to divides on the right-hand side. For the induction hypothesis, the substitution for induction variables is $x = u, y = v$, whereas the substitution for the conclusion is $x = u, y = u + v$ where $v < u + v$.

The first base case leads to: $\mathsf{divides}(u, 0+0)$ if $\mathsf{divides}(u, 0) \wedge u > 0$, which simplifies to true using the definition of $+$. The second base case is $\mathsf{divides}(u, v+v)$ if $\mathsf{divides}(u, v) \ \wedge \ (u > 0) \wedge (v < u) \ \wedge \ v \neq 0$, which trivially simplifies to true since $\mathsf{divides}(u, v)$ in the condition simplifies to false under the condition $v < u \ \wedge \ v \neq 0$. The induction step case is:

$$\mathsf{divides}(u, (u+v)+(u+v)) \text{ if } \mathsf{divides}(u, u+v) \wedge u > 0 \wedge (v < u+v),$$

with the induction hypothesis being: $\mathsf{divides}(u, v+v)$ if $\mathsf{divides}(u, v) \wedge u > 0$. Using the properties of $<$, the associativity and commutativity properties of $+$ and the definition of divides, the conclusion above reduces to: $\mathsf{divides}(u, v+v)$ if $\mathsf{divides}(u, v) \wedge u > 0$, the induction hypothesis. So the conjecture is proved.

The reader would have noticed how using the well-founded ordering suggested by the definition of divides leads to an induction hypothesis which is useful in proving the conjecture. Also observe a close connection between the rewrite rules defining divides and the induction scheme generated for x, y, the arguments to divides in the conjecture. The cover set method discussed below formalizes these ideas. Also, the reader would have noticed the role of a decision procedure for Presburger arithmetic in determining whether divides is completely defined and in simplifying subgoals. This will be further elaborated in later sections.

2. Preliminaries and Background

We use many-sorted first-order logic with equality ($=$) as the framework for discussing proofs. For a signature $\mathcal{F}$ and an infinite set of variables $\mathcal{V}$, let $\mathcal{T}erms(\mathcal{F},\mathcal{V})$ and $\mathcal{T}erms(\mathcal{F})$ denote, respectively, the set of (well-typed) *terms* and the set of *ground terms* (terms without variables) over $\mathcal{F}$. Below, by x^*, we mean a k-tuple of pairwise different variables $x_1,\ldots,x_k$, where k depends upon the context; similarly, s^* stands for a k-tuple of terms $s_1,\ldots,s_k$.

A position is a sequence of positive integers used to refer to a subterm in a term. An equation will be considered as a term with $=$ as the binary predicate. A conditional equation will be considered as a term with $=$ as the binary predicate whose second argument is a term with the outermost function symbol "if" considered as a binary function. In the above conjecture about divides, for instance, the position of $\mathsf{divides}(x,y)$ is 2.2.1 as the conjecture is viewed as an abbreviation for $\mathsf{divides}(x,y+y) = (true \text{ if } \mathsf{divides}(x,y) \wedge x > 0)$.

2.1. Decidable Theories Capturing the Semantics of Data Structures

To incorporate the semantics of data structures on which recursive function definitions are given, let $\mathcal{T}$ stand for a theory associated with a recursive data structure. A theory $\mathcal{T}$ is given by a finite signature $\mathcal{F}_{\mathcal{T}}$ and a set of axioms $AX_{\mathcal{T}}$ over the signature $\mathcal{F}_{\mathcal{T}}$. The theory $\mathcal{T}$ is defined to be the set of all closed formulas φ over $\mathcal{F}_{\mathcal{T}}$ such that $AX_{\mathcal{T}} \models \varphi$ (then φ is called *valid*). Here, "$\models$" is the usual (semantic) first-order consequence relation. We often omit leading universal quantifiers and we write $s =_{\mathcal{T}} t$ as a shorthand for $AX_{\mathcal{T}} \models \forall \ldots s = t$. A term built from variables and $\mathcal{F}_{\mathcal{T}}$ is called a $\mathcal{T}$-term.

We mostly use two theories in this paper for illustration: The theory of free constructors $\mathcal{T}_C$ and Presburger arithmetic $\mathcal{T}_{PA}$. The theory of free constructors $\mathcal{T}_C$ has a finite set of free constructors in $\mathcal{F}_{\mathcal{T}_C}$. Axioms $AX_{\mathcal{T}_C}$ for the *theory $\mathcal{T}_C$ of free constructors* are:

$\neg c(x_1,\ldots,x_n) = c'(y_1,\ldots,y_m)$	for all $c, c' \in \mathcal{F}_{\mathcal{T}_C}$ where $c \neq c'$
$c(x_1,..,x_n) = c(y_1,..,y_n) \Rightarrow x_1 = y_1 \wedge ... \wedge x_n = y_n$	for all $c \in \mathcal{F}_{\mathcal{T}_C}$
$\bigvee_{c\in\mathcal{F}_{\mathcal{T}_C}} \exists y_1,\ldots,y_m \;\; x = c(y_1,\ldots,y_m)$	
$\neg\ (c_1(\ldots c_2(\ldots c_n(\ldots x\ldots)\ldots)\ldots) = x)$	for all sequences $c_1,...,c_n$, $c_i \in \mathcal{F}_{\mathcal{T}_C}$

Note that the last type of axioms usually results in infinitely many formulas. Here, "$\ldots$" in the arguments of c_i stands for pairwise different variables.

For natural numbers, for example, the constructors are $\mathsf{0}$ and s (successor), which are free; that means that (i) $\mathsf{0}$ is assumed to stand for a number different from a number represented by $\mathsf{s}(x)$, and (ii) if $\mathsf{s}(x)$ and $\mathsf{s}(y)$ denote the same number, then x and y also denote the same number. Moreover, (iii) every number is either $\mathsf{0}$ or of the form $\mathsf{s}(x)$, and (iv) $\mathsf{s}^n(x)$ is different from x for all $n \geq 1$.

There is yet another way to characterize the theory of natural numbers using *Presburger Arithmetic*, which is richer. We use the following definition for the *theory $\mathcal{T}_{PA}$*: $\mathcal{F}_{\mathcal{T}_{PA}} = \{0, 1, +\}$ and $AX_{\mathcal{T}_{PA}}$ consists of the following formulas:

$$\begin{array}{rcl@{\qquad\qquad}rcl} (x+y)+z & = & x+(y+z) & \neg\ (1+x & = & 0) \\ x+y & = & y+x & x+y=x+z & \Rightarrow & y=z \\ 0+y & = & y & x=0 & \vee & \exists y\ \ x=y+1 \end{array}$$

For $t \in \mathcal{T}erms(\mathcal{F}_{\mathcal{T}_{PA}},\mathcal{V})$ with $\mathcal{V}(t) = \{x_1,\ldots,x_m\}$, there exist $a_i \in \mathbb{N}$ such that $t =_{\mathcal{T}_{PA}} a_0 + a_1 \cdot x_1 + \ldots + a_m \cdot x_m$. Here, "$a \cdot x$" denotes the term $x + \ldots + x$ (a times) and "a_0" denotes $1 + \ldots + 1$ (a_0 times). We often write *flattened* terms (i.e., without parentheses) since "$+$" is associative and commutative. For $s =_{\mathcal{T}_{PA}} b_0 + b_1 \cdot x_1 + \ldots + b_m \cdot x_m$ and t as above, we have $s =_{\mathcal{T}_{PA}} t$ iff $a_0 = b_0,\ldots,a_m = b_m$.

The theory $\mathcal{T}_{PA}$ is an example where constructors are not free, i.e., there are nontrivial equalities between constructor terms. Similarly, for integers, the constructors $\mathsf{0}$, s, and p (predecessor) are not free since $\mathsf{0} = \mathsf{s}(\mathsf{p}(\mathsf{0})) = \mathsf{p}(\mathsf{s}(\mathsf{0}))$. If constructors are not free, then relations over constructor symbols are usually assumed to be expressed using equations.

2.2. Recursive Function Definitions

A function definition is assumed to be given using a finite set of terminating rewrite rules. To be precise, we use *term rewrite systems* (TRSs) over a signature $\mathcal{F} \supseteq \mathcal{F}_\mathcal{T}$ as our programming language [2] and require that all left-hand sides of rules have the form $f(s^*)$ for a tuple of terms s^* from $\mathcal{T}erms(\mathcal{F}_\mathcal{T}, \mathcal{V})$ and $f \notin \mathcal{F}_\mathcal{T}$. Let $\mathcal{F}_d = \mathcal{F} \setminus \mathcal{F}_\mathcal{T}$ denote the set of *defined symbols*. For a defined function f, D_f denotes the rewrite rules defining f.

To perform evaluations with the TRS $\mathcal{R}$ and the underlying theory $\mathcal{T}$, we use rewriting modulo a theory, where $\rightarrow_{\mathcal{R}/\mathcal{T}}$ must be decidable (e.g., this holds if $\mathcal{T}$-equivalence classes of terms are finite and computable). We have $s \rightarrow_{\mathcal{R}/\mathcal{T}} t$ iff there are s' and t' with $s =_\mathcal{T} s' \rightarrow_\mathcal{R} t' =_\mathcal{T} t$.

Unless stated otherwise, we restrict ourselves to terminating, confluent, and sufficiently complete TRSs $\mathcal{R}$, where $\mathcal{R}$ is *terminating* iff $\rightarrow_{\mathcal{R}/\mathcal{T}}$ is well founded, it is *confluent* iff $\rightarrow_{\mathcal{R}/\mathcal{T}}$ is confluent, and it is *sufficiently complete* iff for all (well-typed) ground terms $t \in \mathcal{T}erms(\mathcal{F})$ there exists a $q \in \mathcal{T}erms(\mathcal{F}_\mathcal{T})$ such that $t \rightarrow^*_{\mathcal{R}/\mathcal{T}} q$ (i.e., q is a *normal form* of t, written as $t{\downarrow_{\mathcal{R}/\mathcal{T}}}$).[2] When regarding $\rightarrow^*_{\mathcal{R}/\mathcal{T}}$ and${\downarrow_{\mathcal{R}/\mathcal{T}}}$, we usually do not distinguish between terms that are equal w.r.t. $=_\mathcal{T}$.

Definition 1 (Inductive Positions) *For $f \in \mathcal{F}_d$, a position i with $1 \leq i \leq arity(f)$ is* non-inductive *if for all f-rules $f(s_1, \ldots, s_m) \rightarrow C[f(t^1_1, \ldots, t^1_m), \ldots, f(t^n_1, \ldots, t^n_m)]$ where C is a context without any occurrence of f, we have $s_i \in \mathcal{V}$, $t^k_i = s_i$, and $s_i \notin \mathcal{V}(s_j) \cup \mathcal{V}(t^k_j)$ for all $j \neq i$ and $1 \leq k \leq n$. Otherwise, the position is* inductive.

In the above definitions of $+, *$, and exponentiation, the second argument is an inductive position, whereas in divides, both arguments are inductive positions. Many induction provers, including *RRL*, use terminating recursive function definitions to generate induction schemes for attempting induction proofs [4, 28, 7, 1].

2.3. Inductive Validity

Instead of *validity*, we are usually interested in *inductive* validity.

Definition 2 (Inductive Validity) *A universal formula $\forall x^* \ \varphi$ is inductively valid in the theory $\mathcal{T}$ (denoted $AX_\mathcal{T} \models_{ind} \varphi$) iff $AX_\mathcal{T} \models \varphi\sigma$ for all ground substitutions σ, i.e., σ substitutes all variables of φ by ground terms of $\mathcal{T}erms(\mathcal{F}_\mathcal{T})$.*

In general, validity implies inductive validity, but not vice versa. We restrict ourselves to theories like $\mathcal{T}_C$ and $\mathcal{T}_{PA}$ which are decidable and inductively complete (i.e., inductive validity of an equation $r_1 = r_2$ (*over* $\mathcal{F}_\mathcal{T}$) also implies its validity, cf. e.g. [9]). Then inductive validity of $r_1 = r_2$ can be checked by a decision procedure for $\mathcal{T}$. Of course, validity and inductive validity do no longer coincide if we introduce additional function symbols defined by rewrite rules.

The rules in $\mathcal{R}$ are considered as equational axioms extending the underlying theory $\mathcal{T}$. This results in a new theory with the signature $\mathcal{F}$ and the axioms $AX_\mathcal{T} \cup \{l = r \mid l \rightarrow r \in \mathcal{R}\}$. To ease readability, we write $AX_\mathcal{T} \cup \mathcal{R}$ instead of $AX_\mathcal{T} \cup \{l = r \mid l \rightarrow r \in \mathcal{R}\}$. It turns out that this extension is *conservative*, i.e., it does not change inductive validity of equations over $\mathcal{F}_\mathcal{T}$.

Theorem 1 *[12] For all $r_1, r_2 \in \mathcal{T}erms(\mathcal{F}_\mathcal{T}, \mathcal{V})$, $AX_\mathcal{T} \models_{ind} r_1 = r_2$ iff $AX_\mathcal{T} \cup \mathcal{R} \models_{ind} r_1 = r_2$.*

3. Cover Sets and Generation of Induction Schemes

The cover set method for mechanizing induction uses terminating function definitions for generating an induction scheme for a data structure. Induction is performed using a well-founded rewrite relation induced

[2] We do however allow recursive function definitions which are not sufficiently complete; for instance, the definition of divides above is not complete.

by the terminating rewrite rules. Two different functions defined differently on the same data structure can give rise to different induction schema.

The left-hand sides of rewrite rules in a definition of f are used to generate different subgoals of an inductive proof of a conjecture in which f occurs. Variables in a conjecture corresponding to the inductive positions in the definition of f are the induction variables which are instantiated by arguments in the left-hand sides at inductive positions. For generating induction hypotheses, any (and all) smaller instantiation(s) in the rewrite relation can be used. One convenient way to generate smaller instantiations for induction hypothesis(es) is to use recursive calls in the right-hand side of the rewrite rules in the definition of f.

Definition 3 (Cover Set) *Let $f \in \mathcal{F}_d$. Its* cover set *is $\mathcal{C}_f = \{\langle s^*, \{t_1^*, \ldots, t_n^*\}\rangle \mid f(s^*) \to C[f(t_1^*), \ldots, f(t_n^*)] \in \mathcal{R}\}$. Similarly, if the rules defining f are conditional rewrite rules, then $\mathcal{C}_f = \{\langle s^*, \{t_1^*, \ldots, t_n^*\}, \{cond_1, \ldots, cond_k\}\rangle \mid f(s^*) \to C[f(t_1^*), \ldots, f(t_n^*)]$ if $cond_1 \wedge \ldots \wedge cond_k \in \mathcal{R}\}$.*

In the above, the second component of a tuple in $\mathcal{C}_f$ is empty if there are no recursive calls to f in the right-hand side of a rule; similarly, the third component of a tuple is empty if the rule is unconditional; $cond$ stands for the set of all literals in the (conjunctive) condition of a conditional rewrite rule. In case a function definition is given using unconditional rewrite rules, its cover set will consist of 2-tuples obtained by dropping the condition.

The cover set based on the definitions of $+$, $*$, and exponentiation is $\mathcal{C}_+ = \mathcal{C}_* = \mathcal{C}_{x^y} = \{\langle\langle x, 0\rangle, \varnothing\rangle, \langle\langle x, s(y)\rangle, \{\langle x, y\rangle\}\rangle\}$; similarly, $\mathcal{C}_{\mathsf{divides}}$: $\{\langle\langle u, 0\rangle, \varnothing, \varnothing\rangle, \langle\langle u, v\rangle, \varnothing, \{v < u, v \neq 0\}\rangle, \langle\langle u, u+v\rangle, \{\langle u, v\rangle\}, \{v < u + v\}\rangle\}$.

3.1. Generating a Basic Induction Scheme from a Cover Set

An induction scheme consists of a finite sets of tuples, in which each tuple has

1. a substitution σ_c for induction variables which is used to generate the conclusion and the associated condition $cond_c$ when the substitution is applicable, as well as

2. a finite (possibly empty) set $\{\sigma_1, \ldots, \sigma_n\}$ of substitutions used to generate induction hypotheses, each associated with a condition $cond_i$ ($1 \leq i \leq n$) when the substitution is applicable, such that

3. each substitution σ_i is smaller in a well-founded reduction ordering than the substitution σ_c.

Each such tuple generates a subgoal of an induction proof attempt. Def. 4 describes how an induction scheme can be generated from a cover set $\mathcal{C}_f$ of f. Here, $\{x^* \to s^*\}$ stands for the substitution $\{x_1 \to s_1, \ldots, x_m \to s_m\}$.

Definition 4 (Basic Induction Scheme) *Given a conjecture in which $f(x^*)$ appears as a subterm, the induction scheme is a finite set of tuples generated as follows: for each cover set triple $\{\langle s^*, \{t_1^*, \ldots, t_n^*\}, \{cond\}\rangle \in \mathcal{C}_f$, there is a tuple $\langle\langle \sigma_c, cond_c\rangle, \{\langle \sigma_i, cond_i\rangle \mid 1 \leq i \leq n\}\rangle$ in the induction scheme, where $\sigma_c = \{x^* \to s^*\}$, $cond_c = cond$, $\sigma_i = \{x^* \to t^*\}$, and $cond_i = cond$.*[3]

The substitution σ_c and each of the substitutions σ_i are linked through the variables shared among the left-hand side and the recursive calls on the right-hand side of the rule from which the cover set tuple is derived.

[3]Sometimes it is helpful to include in the induction scheme, the position of the subterm $f(x^*)$ in the conjecture as well as its replacement both for the conclusion as well as each of the induction hypotheses; see [17] for more details.

3.2. Induction Schemes from Nonbasic Subterms of a Conjecture

Sometimes an induction scheme corresponding to a nonbasic subterm (i.e., different from $f(x^*)$) in a conjecture has to be generated. This could be because either there is no basic subterm in the conjecture, variables in the conjecture on which induction must be performed to get a proof, appear only as arguments in a nonbasic subterm, or the induction scheme generated from a basic subterm does not produce a proof. We give below an algorithm for generating an induction scheme for a nonbasic subterm $f(r^*)$ from the cover set $\mathcal{C}_f$ of f. Assume below that variables in a conjecture (including $f(r^*)$) are disjoint from variables used in $\mathcal{C}_f$.

An induction scheme generated for $f(r^*)$ using the cover set $\mathcal{C}_f$ is a finite set of tuples generated as follows. Here, $r^* = s^*$ stands for $r_1 = s_1 \wedge \ \ldots \ \wedge r_k = s_k$. For each tuple $\{\langle s^*, \{t_1^*, \ldots, t_n^*\}, \{cond\}\rangle \in \mathcal{C}_f$, we include the tuple $\langle\langle \sigma_c, cond_c\rangle, \{\langle \sigma_i, cond_i\rangle \mid 1 \leq i \leq n\}\rangle$ in the induction scheme, where

1. σ_c is a most general unifier of $r^* = s^*$ in $\mathcal{T}$ under the constraint $cond$, possibly generating a feasibility constraint c_c on variables in $\mathcal{C}_f$ under which σ_c is applied; $cond_c = \sigma_c(cond) \wedge c_c$;

2. each σ_i is a most general unifier of $r^* = \sigma_c(t_i^*)$ in $\mathcal{T}$ under the feasibility constraint c_c from the previous step, also generating a feasibility constraint c_i, and $cond_i = \sigma_i(cond) \wedge c_i$.

3. In case of multiple most general unifiers, each must be considered.

Consider the following definition of the greatest common divisor (gcd).

1. $\mathsf{gcd}(0, x) \rightarrow x$,
2. $\mathsf{gcd}(x, 0) \rightarrow x$,
3. $\mathsf{gcd}(x + y, y) \rightarrow \mathsf{gcd}(x, y)$ if $x + y > x$,
4. $\mathsf{gcd}(x, x + y) \rightarrow \mathsf{gcd}(x, y)$ if $x + y > y$.

We obtain the cover set $\mathcal{C}_{\mathsf{gcd}} = \{\langle\langle 0, x\rangle, \varnothing, \varnothing\rangle, \langle\langle x, 0\rangle, \varnothing, \varnothing\rangle, \langle\langle x + y, y\rangle, \{\langle x, y\rangle\}, \{x + y > x\}\rangle, \langle\langle x, x + y\rangle, \{\langle x, y\rangle\}, \{x + y > y\}\rangle\}$; the conditions are included in the recursive rules to ensure termination of rewriting.

The induction scheme for the conjecture $\mathsf{gcd}(m + m, 2) = 2$ (where 2 stands for $1 + 1$) is generated as follows. The base cases are obtained from the first two tuples in the cover set. First, (i) $\{m + m = 0, 2 = x\}$ are unified w.r.t. $\mathcal{T}_{PA}$, giving the substitution $\{m \rightarrow 0, x \rightarrow 2\}$; the corresponding subgoal is $\mathsf{gcd}(0 + 0, 2) = 2$; (ii) $\{m + m = x, 2 = 0\}$ has no solution, so no base case is generated.

The induction step case from the third tuple in the cover set is generated as follows. For the conclusion, we solve $\{m+m = x+y, 2 = y\}$ under the constraint $x+y > x$ giving the substitution $\{m \rightarrow z+1, y \rightarrow 2, x \rightarrow 2\cdot z\}$. Here, $2\cdot z$ again stands for $z+z$. For the induction hypothesis, $\{m+m = 2\cdot z, 2 = 2\}$ is solved to get the substitution $\{m \rightarrow z\}$. Thus the subgoal generated is: $\mathsf{gcd}(z+z, 2) = 2 \Rightarrow \mathsf{gcd}((z+z)+2, 2) = 2$.

Similarly, the induction step case from the fourth tuple of the cover set is obtained as follows: $\{m+m = x, 2 = x + y\}$ is solved under the constraint $x + y > y$ giving the substitution $\{m \rightarrow 1, y \rightarrow 0, x \rightarrow 2\}$ for the conclusion. The unification problem $\{m + m = 2, 2 = 0\}$ has no solution; so there is no induction hypothesis for this case. Thus the subgoal generated is: $\mathsf{gcd}(1 + 1, 2) = 2$. Many other examples of how induction schemes are generated from nonbasic terms are included in [17].

Using the above generated induction scheme, the conjecture can be easily proved as each subgoal is trivial to establish. In contrast, such conjectures are usually proved with other theorem provers using hints and other tricks (see for instance, proofs of such properties in [4, 5]).

3.3. Completeness of Cover Sets and Induction Schemes

An induction scheme used for attempting a proof by induction must be both *sound* and *complete*. The completeness and soundness of an induction scheme are directly linked to the properties of the underlying cover set from which it is obtained. The soundness is guaranteed if the induction scheme (in particular induction hypotheses) is generated from a terminating function definition: recursive calls in the right-hand

side in a function definition are guaranteed to be smaller than the left-hand side of each rule in some well-founded reduction ordering used to prove the termination of the function definition. The reduction ordering must also preserve equivalence $=_{\mathcal{T}}$ w.r.t. $\mathcal{T}$.

To ensure the completeness of an induction scheme, the cover set of a function f used to generate it must provide a *complete cover* for the domain of f. In other words, D_f must be completely defined, i.e., for all $q_1, \ldots, q_k \in \mathcal{T}erms(\mathcal{F}_{\mathcal{T}})$ there must be a $q \in \mathcal{T}erms(\mathcal{F}_{\mathcal{T}})$ such that $f(q_1, \ldots, q_k) \rightarrow^* q$.

Theorem 2 *An induction scheme generated from a term $f(r_1, \ldots, r_k)$ using the cover set $\mathcal{C}_f$ is complete and sound if D_f is complete.*

3.3.1. A Formula Characterizing the Coverage by a Function Definition

Below, the method using a decision procedure for a theory ($\mathcal{T}_C$ or $\mathcal{T}_{PA}$) proposed in [13] to check for completeness of a function definition is reviewed.

For each rewrite rule in the definition D_f of f, a formula can be constructed specifying the subset of the domain of f covered by the rewrite rule. The subset of the domain covered by D_f is then the disjunction of the formulas corresponding to each rewrite rule. Let $x_1, \ldots, x_k$ be new variables denoting the arguments of f. Corresponding to the left-hand side, say $l = f(s^*)$ of each rule "$l \rightarrow r$ if $cond$", we construct the *domain formula* $\exists u^* \ (x^* = s^* \ \wedge \ cond)$, where u^* are the variables appearing in s^*. The above formula is true for those values of x^* for which substitutions for u^* can be found such that $x^* = s^*$ and $cond$ is satisfied. Let Φ_f stand for the disjunction of the domain formulas of all rules in the definition D_f. If Φ_f is valid in $\mathcal{T}$, then D_f is complete.

For the definition of exponentiation, for example, the above method produces the domain formula:

$$\exists u_1, u_2 \ [(x = u_1 \wedge y = 0) \vee (x = u_1 \wedge y = \mathsf{s}(u_2))]$$

over the theory of free constructors 0 and s. This formula is clearly valid, implying that the definition of exponentiation and the associated cover set are complete. Similarly, it can be shown that the formula corresponding to the coverage of the definition of gcd is valid in $\mathcal{T}_{PA}$.

¿From the definition of divides, Φ is:

$$\exists u \ (x_1 = u \wedge x_2 = 0) \vee \exists u, v \ (x_1 = u \wedge x_2 = v \wedge v < u \wedge v \neq 0)$$

$$\vee \exists u, v \ (x_1 = u \wedge x_2 = u + v \wedge v < u + v).$$

The above formula can be shown to be not valid, implying that the definition of divides is incomplete: divides is defined only if $x_1 = \mathsf{s}(u_1) \vee x_2 = 0$ and not defined if $x_1 = 0 \wedge x_2 = \mathsf{s}(u_2)$. This equivalent quantifier-free formula can be obtained using a quantifier elimination procedure [9] on Φ.

In [15], algorithms for checking completeness of constructor-based definitions expressed as terminating rewrite rules are discussed; constructors are assumed either to be free (with no relations) or relations on constructor terms are specified as a convergent rewrite system.

3.3.2. Incomplete Cover Sets and Associated Induction Schemes

If a cover set is incomplete, unsound conclusions can be made unless one is careful in using an induction scheme generated from it. For example, it is shown above that the cover set based on the definition of divides is incomplete. Using an induction scheme generated from it, it is possible to conclude that

$$\mathsf{divides}(x, y + y) \ \text{ if } \ \mathsf{divides}(x, y),$$

even though the terms in the above formula are not defined for all values of x and y.

Given a conjecture φ, a proof of $\Phi_f \ \Rightarrow \ \varphi$ is attempted if the induction scheme generated from the cover set of f is used, where Φ_f characterizes the subdomain covered by the definition of f. If $\Phi_f \ \Rightarrow \ \varphi$ is equivalent to φ, a proof of φ is being attempted as the example below illustrates.

Since divides is defined only if $x > 0 \vee y = 0$, the above conjecture may only be formulated under this condition. Thus,

$$\mathsf{divides}(x, y + y) \text{ if } \mathsf{divides}(x, y) \wedge (x > 0 \vee y = 0)$$

is a theorem. Similarly,

$$\mathsf{divides}(2, x) = \mathsf{not}(\mathsf{divides}(2, \mathsf{s}(x))) \text{ if } (2 > 0 \vee x = 0)$$

can be proved using induction; this conjecture is the same as

$$\mathsf{divides}(2, x) = \mathsf{not}(\mathsf{divides}(2, \mathsf{s}(x))),$$

since the condition in the conditional formula is true.

3.4. Merging of Induction Schemes

Induction schemes suggested by different subterms of a given conjecture can in general share induction variables amongst each other. An inductive proof attempt of the conjecture based on one of these schemes only is not likely to succeed in such cases. The following example illustrates this. Consider the conjecture

$$\mathsf{same}(u, w) \text{ if } \mathsf{same}(u, v) \ \wedge \ \mathsf{same}(v, w),$$

where same is defined on *finite lists* with free constructors nil and cons:

1. $\mathsf{same}(\mathsf{nil}, \mathsf{nil}) \rightarrow \mathsf{true}$, 2. $\mathsf{same}(\mathsf{nil}, \mathsf{cons}(x_1, y_1)) \rightarrow \mathsf{false}$,
3. $\mathsf{same}(\mathsf{cons}(x_1, y_1), \mathsf{nil}) \rightarrow \mathsf{false}$,
4. $\mathsf{same}(\mathsf{cons}(x_1, y_1), \mathsf{cons}(x_2, y_2)) \rightarrow (x_1 = x_2) \wedge \mathsf{same}(y_1, y_2)$.

The induction schemes suggested by $\mathsf{same}(u, v)$, $\mathsf{same}(v, w)$, and $\mathsf{same}(u, w)$ are three possible candidates for attempting a proof of the above conjecture by induction. The schemes are similar; they differ in the induction variables.

Attempting a proof of the above conjecture using the scheme generated from $\mathsf{same}(u, v)$ would result in the induction step case with the conclusion,

$$\mathsf{same}(\mathsf{cons}(x_1, y_1), w) \text{ if } \mathsf{same}(\mathsf{cons}(x_1, y_1), \mathsf{cons}(x_2, y_2)) \wedge \mathsf{same}(\mathsf{cons}(x_2, y_2), w),$$

and the hypothesis, $\mathsf{same}(y_1, w)$ if $\mathsf{same}(y_1, y_2) \wedge \mathsf{same}(y_2, w)$. The conclusion simplifies using the last rule in the definition of same to,

$$\mathsf{same}(\mathsf{cons}(x_1, y_1), w) \text{ if } (x_1 = x_2) \wedge \mathsf{same}(y_1, y_2) \wedge \mathsf{same}(\mathsf{cons}(x_2, y_2), w).$$

The hypothesis does not match the conclusion and the proof attempt by induction fails. [4] The failure is due to w not being instantiated in $\mathsf{same}(u, w)$ and $\mathsf{same}(v, w)$. Similarly, a proof attempt based on the other two schemes would also get stuck at a similar subgoal since one of u or v would remain uninstantiated.

This situation can be remedied if the induction schemes due to $\mathsf{same}(u, v)$ and $\mathsf{same}(v, w)$ are merged; this amounts to performing simultaneous induction on all the three variables of the conjecture. For the above example, each of u, v, w is instantiated to be one of $\mathsf{nil}, \mathsf{cons}(x_i, y_i)$. Further, merging of schemes also eliminates the need to arbitrarily choose from amongst competing schemes.

In general, if two subterms $t_1 = f(x, y)$ and $t_2 = g(z, x)$ in a given conjecture where both the arguments of f, g are inductive positions, share a common variable, x, then attempting a proof of the conjecture by induction based only on the scheme suggested by t_1 would generate $t_1' = \sigma_c(f(x, y)) = f(\sigma_c(x), \sigma_c(y))$ and $t_2' = \sigma_c(g(z, x)) = g(z, \sigma_c(x))$ in the conclusion of an induction step case which cannot be simplified

[4] A proof of the above simplified subgoal can be successfully attempted by induction again possibly using an induction scheme generated from the non-basic term $\mathsf{same}(\mathsf{cons}(x_1, y_1), w)$.

to match the induction hypothesis. The same will be true if the induction scheme suggested by t_2 is used. Instead, the induction schemes from t_1, t_2 need to be merged.

Given a substitution $\langle\langle\sigma_c, cond_c\rangle, \{\langle\sigma_{h_i}, cond_{h_i}\rangle\}\rangle$ from one induction scheme and another substitution $\langle\langle\theta_c, cond'_c\rangle, \{\langle\theta_{h_i}, cond'_{h_i}\rangle\}\rangle$ from another induction scheme for a conjecture, if they share a variable, then the substitutions for the variable in σ_c and θ_c are reconciled by unification. Let δ be a most general unifier of $\sigma_c(x)$ and $\theta_c(x)$, where x is a shared variable. Then the merged substitution is obtained by applying δ on both the substitutions and merging them in an obvious way, as well as taking a conjunction of the instantiated conditions. In case of more than one mgu, each mgu gives a case for the merged induction scheme. If two induction schemes are merged, the result can sometimes be an induction scheme equivalent to one of the original schemes except for additional induction hypotheses which are often useful in completing a proof attempt that may otherwise get stuck. For examples and details, we refer to [17].

4. Automatically Deciding Equations by Induction

In the late 1990s, Kapur and Subramaniam proposed integrating induction into decision procedures without compromising automation [22]. Particularly, we are interested in identifying structural conditions on recursive function definitions such that a large family of conjectures expressed using such function symbols can be decided automatically using the cover set induction method (of course with the aid of decision procedures). In a series of papers [11, 12, 22], Kapur, Giesl, and Subramaniam have proposed syntactic conditions on recursive function definitions as well as on conjectures which guarantee that inductive validity of these conjectures is decidable. Another motivation for this line of research has been to characterize the class of formulas our theorem prover *RRL* can decide automatically (without any user guidance). In this and the next section, we give an overview of the results. For simplicity, it is assumed that all function definitions are given using terminating unconditional rewrite rules; consequently, the associated cover sets and induction schemes will not have any $cond$ component in the tuples.

4.1. $\mathcal{T}$-based Function Definitions and Compatible Definitions

In [22], we introduced the concept of a $\mathcal{T}$*-based* function definition: in the rewrite rules defining a function symbol f, (i) all arguments to f are $\mathcal{T}$-terms and (ii) the right-hand side becomes a $\mathcal{T}$-term after subterms of the form $f(t^*)$, if any, are abstracted using new variables.

Definition 5 ($\mathcal{T}$-based Definition) *A function symbol $f \in \mathcal{F}$ has a* $\mathcal{T}$-based *definition iff $f \in \mathcal{F}_\mathcal{T}$ or if all rules $l \to r \in \mathcal{R}$ have the form $f(s^*) \to C[f(t_1^*), \ldots, f(t_n^*)]$, where $s^*, t_1^*, \ldots, t_n^*$ are from $\mathcal{T}erms(\mathcal{F}_\mathcal{T}, \mathcal{V})$ and C is a context over $\mathcal{F}_\mathcal{T}$.*

For instance, the definition of $+$ in Section 1.2 is $\mathcal{T}_C$-based with $0, \mathsf{s}$ as the set of free constructors; the definitions of $*$ and exponentiation are however not $\mathcal{T}_C$-based since $+$ and $*$ occur, respectively, in the right-hand sides of these definitions.

The definitions of gcd and divides are $\mathcal{T}_{PA}$-based. In general, definitions like those of gcd and divides, which involve conditions are declared $\mathcal{T}$-based if the function symbols in the conditions belong to $\mathcal{F}_\mathcal{T}$ [22].

4.2. Compatibility among Function Definitions

Now we introduce the concept of *compatible function definitions*: Compatibility ensures that when a term built from a compatible sequence of function symbols is instantiated and normalized, the result is a $\mathcal{T}$-term. Informally, the definition of g is compatible with the definition of f if for every rule of f, the *context* created by the rule on its right-hand side can be simplified by the definition of g.

Formally, the definition of a function g is compatible with the definition of f on argument j if in any term $g(\ldots, f(\ldots), \ldots)$, where f is on the j-th argument of g, every context created by rewriting f will move outside the term by rewriting g. So if f has a rule $f(s^*) \to C[f(t_1^*), \ldots, f(t_n^*)]$, then

rewriting f can create the context C. Hence, if induction on f is performed within a term of the form $g(\ldots f(\ldots)\ldots)$, then in the induction conclusion, the resulting term $g(\ldots f(s^*)\ldots)$ can be rewritten to a term $D[\,g(\ldots f(t^*_{i_1})\ldots),\ \ldots,\ g(\ldots f(t^*_{i_k})\ldots)\,]$. Here, the induction hypotheses $g(\ldots f(t^*_i)\ldots)$ occur within a context D. Consider the following definitions:

$$
\begin{array}{rcl}
\mathsf{mod2}(0) & \to & 0,\\
\mathsf{mod2}(\mathsf{s}(0)) & \to & \mathsf{s}(0),\\
\mathsf{mod2}(\mathsf{s}(\mathsf{s}(x))) & \to & \mathsf{mod2}(x),\\
\mathsf{exp2}(0) & \to & \mathsf{s}(0),\\
\mathsf{exp2}(\mathsf{s}(x)) & \to & \mathsf{exp2}(x)+\mathsf{exp2}(x),\\
\mathsf{log2}(\mathsf{s}(0)) & \to & 0,\\
\mathsf{log2}(x+x) & \to & \mathsf{s}(\mathsf{log2}(x)),\\
\mathsf{log2}(\mathsf{s}(x+x)) & \to & \mathsf{s}(\mathsf{log2}(x)),\\
\mathsf{len}(\mathsf{nil}) & \to & 0,\\
\mathsf{len}(\mathsf{cons}(n,x)) & \to & \mathsf{s}(\mathsf{len}(x)),
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{half}(0) & \to & 0,\\
\mathsf{half}(\mathsf{s}(0)) & \to & 0,\\
\mathsf{half}(\mathsf{s}(\mathsf{s}(x))) & \to & \mathsf{s}(\mathsf{half}(x)),\\
\mathsf{dbl}(0) & \to & 0,\\
\mathsf{dbl}(\mathsf{s}(x)) & \to & \mathsf{s}(\mathsf{s}(\mathsf{dbl}(x))),\\
\mathsf{min}(0,y) & \to & 0,\\
\mathsf{min}(\mathsf{s}(x),0) & \to & 0,\\
\mathsf{min}(\mathsf{s}(x),\mathsf{s}(y)) & \to & \mathsf{s}(\mathsf{min}(x,y)),\\
\mathsf{app}(\mathsf{nil},y) & \to & y,\\
\mathsf{app}(\mathsf{cons}(n,x),y) & \to & \mathsf{cons}(n,\mathsf{app}(x,y)).
\end{array}
$$

The function $+$ is compatible with dbl on the second argument; $+$ is compatible with min and len on the second argument also. The function half is compatible with $\mathsf{mod2}$; however, $\mathsf{mod2}$ is not compatible with half. However, $\mathsf{mod2}$ is compatible with itself but half is not compatible with itself.

Definition 6 (Compatible Functions) *Let g, f be $\mathcal{T}$-based, $f \notin \mathcal{F}_\mathcal{T}$, and $1 \le j \le m = arity(g)$. The definition of g* is compatible *with the definition of f* on argument j *iff for all rules $\alpha : \ f(s^*) \to C[f(t^*_1),\ldots,f(t^*_n)]$, we have*

$$
\begin{array}{l}
g(x_1,\ldots,x_{j-1},C[z_1,\ldots,z_n],x_{j+1},\ldots,x_m) \quad \to^*_{\mathcal{R}/\mathcal{T}}\\
D[\,g(x_1,\ldots,x_{j-1},\,z_{i_1},\,x_{j+1},\ldots,x_m),\ \ldots,\ g(x_1,\ldots,x_{j-1},\,z_{i_k},\,x_{j+1},\ldots,x_m)\,]
\end{array}
$$

for a context D over $\mathcal{F}_\mathcal{T}$, $i_1,\ldots,i_k \in \{1,\ldots,n\}$, $z_i \notin \mathcal{V}(D)$ for all i.

Note that in Def. 6, g can also be a symbol of $\mathcal{F}_\mathcal{T}$. For instance, s is compatible with len.

The concept of compatibility can be extended to arbitrarily deep nesting leading to a *compatibility sequence* [22]. Consider a term $f_1(p^*_1,\ f_2(p^*_2,\ f_3(x^*),\ q^*_2),\ q^*_1)$, where the pairwise different variables x^* on f_3's inductive positions do not occur in the terms p^*_i, q^*_j with f_2 at j_1-th argument of f_1 and f_3 at j_2-th argument of f_2. The definition of compatibility sequences should guarantee that if $\langle f_1, f_2, f_3\rangle$ is a compatibility sequence on the arguments $\langle j_1, j_2\rangle$, then in an induction on f_3, the resulting context is propagated outside of the whole term. Hence, we require that f_i must be compatible with f_{i+1} on argument j_i for all $i \in \{1,2\}$. For example, $\langle \mathsf{s}, \mathsf{len}, \mathsf{app}\rangle$ is a compatibility sequence on $\langle 1,1\rangle$. For more details about compatibility sequences of function definitions as well as *simultaneous compatibility*, the reader should refer to [22] and [12].

Theorem 3 *[22] An equation $f_1(p^*_1, f_2(p^*_2,\ldots f_k(x^*)\ldots,q^*_2),q^*_1) = r$ is decidable using the induction scheme based on the cover set $\mathcal{C}_{f_k}$ of f_k, if r is a $\mathcal{T}$-term, each f_i has a $\mathcal{T}$-based definition, $\langle f_1,\ldots,f_k\rangle$ is a compatibility sequence, and p^*_i, q^*_i are $\mathcal{T}$-terms not containing the variables x^*.*

Up to now, $+$ is not compatible with itself on the second argument because of the non-recursive rule. In [12], we allow the compatibility requirement to be violated for non-recursive rules so that more function definitions can be compatible with each other and our results are more widely applicable. Then, dbl is also compatible with $+$ and len is compatible with app.

The conditions of Theorem 3 can be easily checked syntactically. They ensure that every subgoal in an induction proof attempt generated using the cover set induction method is such that (i) the induction hypothesis(es) generated from the cover set method is applicable and (ii) each subgoal reduces to a formula over $\mathcal{F}_\mathcal{T}$ and can thus be decided. For example, in case of the conjecture $\mathsf{log2}(\mathsf{exp2}(x)) = x$, the base case

obtained by substituting $\mathsf{0}$ for x gives $\mathsf{0} = \mathsf{0}$ which is valid in Presburger arithmetic; for the induction step case, the conclusion is $\mathsf{log2}(\mathsf{exp2}(\mathsf{s}(y))) = \mathsf{s}(y)$ which simplifies to $\mathsf{s}(\mathsf{log2}(\mathsf{exp2}(y))) = \mathsf{s}(y)$. Then the induction hypothesis $\mathsf{log2}(\mathsf{exp2}(y)) = y$ applies, giving $\mathsf{s}(y) = \mathsf{s}(y)$ which is valid in Presburger arithmetic. Examples of conjectures which can be decided in this way are:

$$\begin{array}{rcl@{\qquad\qquad}rcl} \mathsf{exp2}(\mathsf{log2}(x)) & = & x, & \mathsf{log2}(\mathsf{exp2}(x)) & = & x, \\ \mathsf{bton}(\mathsf{pad0}(\mathsf{ntob}(x))) & = & x, & \mathsf{last}(\mathsf{ntob}(\mathsf{dbl}(x))) & = & \mathsf{0}. \end{array}$$

Functions bton, ntob convert bit vectors to numbers and vice versa, respectively; $\mathsf{pad0}$ pads zeros to a bit vector; last gives the last bit of the bit vector output by ntob; for definitions, the reader may consult [21].

4.3. Equational Conjectures with Defined Symbols on Both Sides

In [12], Giesl and Kapur relaxed the restrictions on the forms of conjectures where inductive validity is decidable. Both sides of an equational conjecture can now contain defined function symbols which have $\mathcal{T}$-based definitions.

Consider a simple conjecture which falls outside the conjectures discussed in the previous section: $\mathsf{dbl}(u + v) = \mathsf{dbl}(u) + v$. During an attempt to prove it by induction on v, the formula $\mathsf{dbl}(u + x) = \mathsf{dbl}(u) + x \;\Rightarrow\; \mathsf{dbl}(u + \mathsf{s}(x)) = \mathsf{dbl}(u) + \mathsf{s}(x)$ in the induction step case simplifies to: $\mathsf{s}(\mathsf{s}(\mathsf{dbl}(u) + x)) = \mathsf{s}(\mathsf{dbl}(u) + x)$. Since it contains the defined symbols $+$ and dbl, its function symbols are not from $\mathcal{F}_{\mathcal{T}_C}$. Similarly, let $*$ be defined on PA-terms with the rules $x * \mathsf{0} \to \mathsf{0}$ and $x * (y + 1) \to x + x * y$. In a proof by induction on w of the distributivity law below

$$(u + v) * w = u * w + v * w, \tag{1}$$

the induction step case $(u + v) * x = u * x + v * x \;\Rightarrow\; (u + v) * (x + 1) = u * (x + 1) + v * (x + 1)$ simplifies to a formula with "$*$" (i.e., it is not in PA):

$$(u + v) + (u * x + v * x) = (u + u * x) + (v + v * x). \tag{2}$$

In [12], it was shown that equational conjectures

$$h(\ldots g(\ldots f(x, y)\ldots)\ldots) = h'(\ldots g'(\ldots f'(x, y)\ldots)\ldots),$$

can be decided under certain conditions, if f, g, h, f', g', h' have $\mathcal{T}$-based function definitions and both $\langle h, g, f\rangle$ and $\langle h', g', f'\rangle$ are compatibility sequences. Under certain conditions, on the positions marked with "$\ldots$" one may also have terms with defined symbols, i.e., terms that are no $\mathcal{T}$-terms. Unlike previous constructions in [11, 22], an induction subgoal for such an equational conjecture need not simplify after the application of induction hypotheses to a formula over $\mathcal{F}_{\mathcal{T}}$; instead the simplified formula still has occurrences of defined function symbols. We developed conditions on the original equational conjecture which ensure that every subgoal which would occur in a proof attempt can be *safely* generalized. Then subterms with defined symbols in the subgoal are abstracted by variables, producing a formula over $\mathcal{F}_{\mathcal{T}}$. If the generalized formula is declared valid by a decision procedure for $\mathcal{T}$, then of course, the original conjecture which is an instance of this generalized formula is valid as well; however, for *safe* generalizations the converse is also true, i.e., if the generalized formula is declared false by the decision procedure, then the original conjecture is false as well. Using these ideas, it becomes possible to syntactically check all the necessary conditions by only inspecting the original conjecture without performing a proof attempt. This results in a decidable class of formulas whose inductive validity can be decided. Some conjectures in this class are:

$$\begin{array}{rcl@{\qquad\qquad}rcl} \mathsf{dbl}(u + v) & = & \mathsf{dbl}(u) + v, & \mathsf{dbl}(u + v) & = & \mathsf{dbl}(u) + \mathsf{dbl}(v), \\ \mathsf{len}(\mathsf{app}(u, v)) & = & \mathsf{len}(u) + \mathsf{len}(v), & \mathsf{s}(\mathsf{len}(\mathsf{app}(u, v))) & = & \mathsf{len}(\mathsf{app}(u, \mathsf{cons}(n, v))), \\ (u + v) * w & = & u * w + v * w, & (u + v) + w & = & u + (v + w), \\ \mathsf{min}(u + w, v + w) & = & \mathsf{min}(u, v) + w. & & & \end{array}$$

4.4. Safe Generalizations by the No-Theory Condition

The requirement of compatible function symbols ensures that in proof attempts, the induction hypothesis will be applicable to the induction conclusion leading to a proof obligation $C[t_1, \ldots, t_n] = D[s_1, \ldots, s_m]$ where C and D are contexts over $\mathcal{F}_\mathcal{T}$ and $t_1, \ldots, t_n, s_1, \ldots, s_m$ are subterms containing defined symbols. These subterms can be determined before the induction proof attempt by inspecting how subterms of the original conjecture get propagated by rewrite rules.

Thus, we need syntactic criteria to ensure that an equation $C[t_1, ..., t_n] = D[s_1, ..., s_m]$ may be generalized to $C[x_{t_1}, ..., x_{t_n}] = D[x_{s_1}, ..., x_{s_m}]$, where t_i, s_j's are subterms with defined function symbols as their roots. Here, t_i and s_j are replaced by fresh variables and identical terms are replaced by the same variable. This generalized equation is an equation over $\mathcal{F}_\mathcal{T}$ and thus, its (inductive) validity can be decided by a decision procedure for $\mathcal{T}$. In general, however, inductive validity of the generalized equation implies inductive validity of the original equation, but not vice versa.

We define a *no-theory* condition which ensures that this generalization is *safe* in the theory of free constructors or Presburger Arithmetic.[5] Then an equation is inductively valid if *and only if* the generalized equation is inductively valid. A term t is said to satisfy the no-theory condition if and only if there is no $\mathcal{T}$-term equivalent to t. Our condition mainly relies on information about the definitions of functions which can be pre-compiled.

Definition 7 (No-Theory) *A term t satisfies the no-theory condition iff there is no $q \in \mathcal{T}erms(\mathcal{F}_\mathcal{T}, \mathcal{V})$ with $AX_\mathcal{T} \cup \mathcal{R} \models_{ind} t = q$. If additionally, $t = f(x^*)$ for pairwise different variables x^*, then f satisfies the no-theory condition, too.*

The no-theory condition is likely to be satisfied for almost all defined functions f (otherwise, the function f is not needed, since one can use the term q instead). It is proved in [12] that the no-theory condition for $\mathcal{T}$-based functions is decidable for $\mathcal{T}_C$ and $\mathcal{T}_{PA}$.

If $f \in \mathcal{F}_d$ does not satisfy the no-theory condition, then there is a term $q \in \mathcal{T}erms(\mathcal{F}_\mathcal{T}, \mathcal{V})$ such that $q[x^*/s^*] =_\mathcal{T} r$ for every non-recursive f-rule $f(s^*) \to r$ (i.e., $r \in \mathcal{T}erms(\mathcal{F}_\mathcal{T}, \mathcal{V})$). In the theory of free constructors, this means that $q[x^*/s^*]$ and r are syntactically identical. Thus, there are only finitely many possibilities for the choice of q. By checking whether these choices for q contradict the remaining rules of f, we can decide the no-theory condition for f. We illustrate the idea with examples.

For $+$ to be expressible in $\mathcal{T}_C$, $x_1 + x_2$ must be equivalent to one of $0, x_1, x_2, \mathsf{s}(t_1)$. From the non-recursive rule $x + 0 \to x$, $x_1 + x_2$ can only be x_1; all other choices are ruled out. But this choice contradicts the second rule $x + \mathsf{s}(y) \to \mathsf{s}(x + y)$: normalizing by $x_1 + x_2 \to x_1$ produces x and $\mathsf{s}(x)$. Similar to $+$, it is easy to see that min, dbl, len, and app satisfy the no-theory condition.

For the theory of Presburger Arithmetic, if $f(x_1, \ldots, x_m) =_{\mathcal{T}_{PA}} q$ where q is a PA-term, then $q =_{\mathcal{T}_{PA}} a_0 + a_1 \cdot x_1 + \ldots + a_m \cdot x_m$ for $a_i \in \mathbb{N}$. Every f-rule $l \to r$ gives a polynomial equation of the form $P_0 + P_1 \cdot y_1 + \ldots + P_k \cdot y_k = Q_0 + Q_1 \cdot y_1 + \ldots + Q_k \cdot y_k$ in the variables $\{y_1, \ldots, y_k\}$ of l (since f is $\mathcal{T}$-based, its rules do not contain nested occurrences of f resulting in linear polynomials). Since they must be equal, we obtain the constraints $P_0 = Q_0, \ldots, P_k = Q_k$. It is decidable whether the set of all these constraints is satisfiable. The constraints are unsatisfiable iff f satisfies the no-theory condition.

For example, for $*$, we assume that $x * y =_{\mathcal{T}_{PA}} a_0 + a_1 \cdot x + a_2 \cdot y$. Its first rule gives $a_0 + a_1 x = 0$; the second rule gives $a_0 + a_1 x + a_2 y + a_2 = a_0 + (a_1 + 1)\ x + a_2 y$. The resulting set of constraints, $\{a_0 = 0, a_1 = 0, a_0 + a_2 = a_0, a_1 = a_1 + 1\}$, is inconsistent. Thus $*$ satisfies the no-theory condition.

Algorithms for checking the no-theory condition for $\mathcal{T}_C$ and $\mathcal{T}_{PA}$ are given in [12], where in addition, the no-theory condition check on a function definition is used to define a no-theory test on terms different from $f(x^*)$.

The following theorem shows that the no-theory condition indeed allows us to replace pairwise variable disjoint terms by fresh variables.

[5] This criterion is generally applicable for safe generalizations. Moreover, one could refine our approach by performing such generalizations also at the beginning before the start of the proof.

Theorem 4 *[12] Let $\mathcal{T}$ be $\mathcal{T}_C$ or $\mathcal{T}_{PA}$ and let $t_1, \ldots, t_n, s_1, \ldots, s_m$ be pairwise identical or variable disjoint terms satisfying the no-theory condition. For all contexts C, D over $\mathcal{F}_{\mathcal{T}}$ and fresh variables x_{t_i} and x_{s_j}, we have $AX_{\mathcal{T}} \cup \mathcal{R} \models_{ind} C[t_1, ..., t_n] = D[s_1, ..., s_m]$ iff $C[x_{t_1}, \ldots, x_{t_n}] =_{\mathcal{T}} D[x_{s_1}, \ldots, x_{s_m}]$.*

The following theorem precisely defines the set of equations with defined function symbols on both sides whose inductive validity is decidable. Using compatibility and book-keeping data structure Pos-sets introduced in [12], it can be estimated which subterms of r_1 and r_2 with defined symbols will occur after simplification without actually attempting an induction proof [12]. Those subterms can be checked for the no-theory condition. Thus, without even doing the induction proof, one can recognize whether the equation will simplify to conjectures over the signature $\mathcal{F}_{\mathcal{T}}$ of the theory. This class called DEC is precisely defined in [12]; an algorithm for deciding equations in DEC is also given. Below, we informally state the main result.

Theorem 5 *[12] An equation $r_1 = r_2$ can be decided using the cover set method, if r_1 and r_2 have compatibility sequences $\langle f_1, \ldots, f_d \rangle$ and $\langle g_1, \ldots, g_e \rangle$, the innermost function symbols on both sides, f_d and g_e, have identical cover sets*[6]*, and the subterms with defined function symbols appearing in the simplified subgoals, which can be computed from the definitions of $f_1, \ldots, f_d$, $g_1 \ldots, g_e$, satisfy the no-theory condition.*

For example, the equations listed at the end of Section 4.3 are in this class DEC and can therefore be decided. For $\mathsf{dbl}(u + v) = \mathsf{dbl}(u) + v$, the left-hand side $\mathsf{dbl}(u + v)$ has the compatibility sequence $\langle \mathsf{dbl}, + \rangle$ and the right-hand side has the compatibility sequence $\langle + \rangle$ with the induction variable v. Using compatibility, one can estimate that only the subterms $\mathsf{dbl}(u)$ and $\mathsf{dbl}(u)+x$ appear in simplified conjectures after application of induction hypotheses. The function dbl satisfies the no-theory condition; therefore, $\mathsf{dbl}(u)$ and $\mathsf{dbl}(u) + x$ also fulfill the no-theory condition. For the distributivity equation $(u + v) * w = u * w + v * w$, the left-hand side has the compatibility sequence $\langle * \rangle$ and the right-hand side has the (simultaneous) sequence $\langle +, (*, *) \rangle$; induction is performed on w. The only subterm with a defined symbol in the simplified conjecture is $u * x + v * x$ which satisfies the no-theory condition.

5. Automatically Deciding Quantifier-free Conjectures

In [11], Giesl and Kapur showed how to extend the class of decidable conjectures from equations to quantifier-free formulas in which equations of the form $h(\ldots, g(\ldots f(x, y) \ldots) \ldots) = r$ are atomic formulas, where $\langle h, g, f \rangle$ forms a compatibility sequence of $\mathcal{T}$-based function definitions and r is a $\mathcal{T}$-term. Some examples of such conjectures include $\mathsf{dbl}(\mathsf{half}(y)) = y \Rightarrow \mathsf{even}(y) = \mathsf{true}$ and $\mathsf{dbl}(y) = y \Rightarrow y = 0$. Equations appearing in these formulas are neither valid nor unsatisfiable; consequently, there is a need to characterize the subset of instantiations for the variables for which these equations are true. For this extension, we need the notion of a *correctness predicate* of an equation.

For simplicity, we will consider $\mathcal{T}$-based definitions in the theory of free constructors $\mathcal{T}_C$; the rewrite system $\mathcal{R}$ consisting of $\mathcal{T}$-based definitions is convergent and sufficiently complete. Furthermore, atomic formulas under consideration are of the form: $r_1 = r_2$, where r_2 is a $\mathcal{T}$-term and if a cover set $\mathcal{C} = \{\langle s_1^*, \{t_{1,1}^*, \ldots, t_{1,n_1}^*\}\rangle, \ldots, \langle s_m^*, \{t_{m,1}^*, \ldots, t_{m,n_m}^*\}\rangle\}$ is used to generate an induction scheme for attempting a proof of such a formula, then we have $r_1[s_i^*] \rightarrow_{\mathcal{R}}^* C_i[r_1[t_{i,1}^*], \ldots, r_1[t_{i,n_i}^*]]$ for a constructor term context C_i for all $1 \leq i \leq m$.

5.1. Correctness Predicates

We present a technique for automatically generating the definition of a *correctness predicate* c_φ for an equation $\varphi : r_1 = r_2$ as a finite set of (unconditional) convergent and sufficiently complete rewrite rules.

[6] This requirement can be weakened by *merging* cover sets, cf. Section 3.4.

For any tuple of constructor ground terms q^*, we have $c_\varphi(q^*) \Rightarrow \varphi[q^*]$. The definition of correctness predicates is similar to the definitions of [10, 26], but its form is quite restricted since we are interested in ensuring that validity of correctness predicates is decidable and that *exact* correctness predicates can be generated which completely characterize the domain of values on which the conjecture holds.

For example, consider the proof of the conjecture $\mathsf{dbl}(\mathsf{half}(y)) = y$ by induction w.r.t. the cover set $\mathcal{C}_{\mathsf{half}}$. If $y = \mathsf{0}$, the conjecture simplifies to true; for $y = \mathsf{s}(\mathsf{0})$, the conjecture reduces to false. In the induction step, where $y = \mathsf{s}(\mathsf{s}(x))$, one has to prove that $\mathsf{dbl}(\mathsf{half}(x)) = x$ implies $\mathsf{dbl}(\mathsf{half}(\mathsf{s}(\mathsf{s}(x)))) = \mathsf{s}(\mathsf{s}(x))$. After simplification and the application of the induction hypothesis, the subgoal reduces to true. This suggests that if the induction hypothesis is valid, then the induction conclusion is also valid.

We thus have the following rules for the correctness predicate $c_{\mathsf{dbl}(\mathsf{half}(y))=y}$:

$$\begin{array}{rcl@{\qquad}rcl} c_{\mathsf{dbl}(\mathsf{half}(y))=y}(\mathsf{0}) & \to & \mathsf{true}, & c_{\mathsf{dbl}(\mathsf{half}(y))=y}(\mathsf{s}(\mathsf{0})) & \to & \mathsf{false}, \\ c_{\mathsf{dbl}(\mathsf{half}(y))=y}(\mathsf{s}(\mathsf{s}(x))) & \to & c_{\mathsf{dbl}(\mathsf{half}(y))=y}(x). & & & \end{array}$$

The above definition is that of the even predicate, which is the condition for $\mathsf{dbl}(\mathsf{half}(y)) = y$ to be valid. Note that the third rule above is stronger than the following rule one would have gotten from the above analysis:

$$c_{\mathsf{dbl}(\mathsf{half}(y))=y}(\mathsf{s}(\mathsf{s}(x))) \quad \to \quad \mathsf{true} \ \text{ if } \ c_{\mathsf{dbl}(\mathsf{half}(y))=y}(x).$$

We will use the third rule above since we wish to synthesize a complete definition of a correctness predicate which is easier to do using unconditional rewrite rules. As a result, the correctness predicate so generated may not be exact (since the third rule would make the correctness predicate false if the recursive call is false on a particular value), and hence, provides only a sufficient condition for the conjecture to be valid.

To prove $r_1 = r_2$ w.r.t. a cover set $\mathcal{C}$, for each pair $\langle s_i^*, \{t_{i,1}^*, \ldots, t_{i,n_i}^*\}\rangle \in \mathcal{C}$, we must check whether the equation $C_i[r_2[t_{i,1}^*], \ldots, r_2[t_{i,n_i}^*]] = r_2[s_i^*]$ obtained after simplification and application of induction hypotheses, is valid. In order to obtain correctness predicates as simple as the ones above, one possibility is to require that such equations are either valid for *all* instantiations or for *none*. This ensures that the right-hand sides of the rules for correctness predicates only have the form true, false, or recursive calls of correctness predicates.

Definition 8 (Radical equations) *An equation $r_1 = r_2$ is* radical *under $\mathcal{C} = \{\langle s_1^*, \{t_{1,1}^*, \ldots, t_{1,n_1}^*\}\rangle, \ldots, \langle s_m^*, \{t_{m,1}^*, \ldots, t_{m,n_m}^*\}\rangle\}$ iff $r_1[s_i^*] \to_{\mathcal{R}}^* C_i[r_1[t_{i,1}^*], \ldots, r_1[t_{i,n_i}^*]]$ for a constructor term context C_i and for each $1 \le i \le m$, either $C_i[r_2[t_{i,1}^*], \ldots, r_2[t_{i,n_i}^*]]] = r_2[s_i^*]$ is valid in $\mathcal{T}_C$ or $\neg C_i[r_2[t_{i,1}^*], \ldots, r_2[t_{i,n_i}^*]]] = r_2[s_i^*]$ is valid in $\mathcal{T}_C$.*

It is easy to see that checking whether an equation is radical is decidable. For instance, the equation $\mathsf{dbl}(\mathsf{half}(y)) = y$ is radical under $\mathcal{C}_{\mathsf{half}}$.

The correctness predicate for a radical equation can be easily synthesized; for a tuple in $\mathcal{C}$ corresponding to a base case, the right-hand side is true or false; for a tuple corresponding to an induction step case, the right-hand side is false or the recursive call on the arguments corresponding to the hypothesis.

To ease the presentation, we will now restrict ourselves to cover sets where there is at most one induction hypothesis for every induction step case.[7] Thus, we only consider cover sets with pairs $\langle s_i^*, \{t_{i,1}^*, \ldots, t_{i,n_i}^*\}\rangle$ where $0 \le n_i \le 1$.

Definition 9 (Correctness Predicate) *Given an equation $r_1 = r_2$ as defined above such that it is* radical *under $\mathcal{C}$, its correctness predicate $c_{r_1=r_2}$ under $\mathcal{C}$ is defined by the following rules:*

$$\begin{array}{rcl} c_{r_1=r_2}(s_i^*) & \to & \begin{cases} \mathsf{true}, & \text{if } \ C_i = r_2[s_i^*] \text{ is valid in } \mathcal{T}_C \text{ and } n_i = 0, \\ \mathsf{false}, & \text{if } \neg C_i = r_2[s_i^*] \text{ is valid in } \mathcal{T}_C \text{ and } n_i = 0, \end{cases} \\ c_{r_1=r_2}(s_i^*) & \to & \begin{cases} c_{r_1=r_2}(t_{i,1}^*), & \text{if } \ C_i[r_2[t_{i,1}^*]] = r_2[s_i^*] \text{ is valid in } \mathcal{T}_C \text{ and } n_i = 1, \\ \mathsf{false}, & \text{if } \neg C_i[r_2[t_{i,1}^*]] = r_2[s_i^*] \text{ is valid in } \mathcal{T}_C \text{ and } n_i = 1. \end{cases} \end{array}$$

[7] The definition of correctness predicates can be easily generalized to the case of multiple induction hypotheses. In fact, correctness predicates can be defined for arbitrary equations.

Theorem 6 *Let $c_{r_1=r_2}$ be a correctness predicate for $r_1 = r_2$ as defined above under $\mathcal{C}$, let $\mathcal{R}$ also contain the rules defining $c_{r_1=r_2}$, let x^* be the variables in $r_1 = r_2$. Then,*

(a) $AX_{\mathcal{T}_C} \cup \mathcal{R} \models_{\text{ind}} c_{r_1=r_2}(x^*) = \mathsf{true} \Rightarrow r_1 = r_2$.

(b) In general, $AX_{\mathcal{T}_C} \cup \mathcal{R} \not\models_{\text{ind}} r_1 = r_2 \Rightarrow c_{r_1=r_2}(x^*) = \mathsf{true}$.

To see that the converse does not hold (b), consider the conjecture $\mathsf{half}(y) = \mathsf{s}(\mathsf{0})$; its correctness predicate $c_{\mathsf{half}(y)=\mathsf{s}(\mathsf{0})}$ is false for the two base cases as well as for the recursive case. However, $\mathsf{half}(y) = \mathsf{s}(\mathsf{0})$ is true for $y = \mathsf{s}^2(\mathsf{0})$ as well as for $y = \mathsf{s}^3(\mathsf{0})$.

5.2. Conjectures with Exact Correctness Predicate

We now characterize equations $r_1 = r_2$ where the correctness predicate $c_{r_1=r_2}$ is *exact*, i.e., for all q^*, $c_{r_1=r_2}(q^*)$ is true iff $AX_{\mathcal{T}_C} \cup \mathcal{R} \models_{\text{ind}} r_1[q^*] = r_2[q^*]$. As stated above, exactness is ensured if in Def. 9, in the case of recursive call, the induction conclusion $r_1[s_i^*] = r_2[s_i^*]$ is equivalent to $r_1[t_{i,1}^*] = r_2[t_{i,1}^*]$. For that reason, the correctness predicate $c_{\mathsf{half}(y)=\mathsf{s}(\mathsf{0})}$ returns false for the arguments $\mathsf{s}^2(\mathsf{0})$ and $\mathsf{s}^3(\mathsf{0})$ although the conjecture is true, since it is false for the smaller arguments $\mathsf{0}$ and $\mathsf{s}(\mathsf{0})$ as well. Thus, $c_{r_1=r_2}$ is only exact if $r_1[q^*] = r_2[q^*]$ implies the validity of $r_1[p^*] = r_2[p^*]$ for all arguments $p^* <_{\mathcal{C}} q^*$ $(*)$, where $<_{\mathcal{C}}$ is the relation described by the cover set $\mathcal{C}$. In the above proof of $\mathsf{dbl}(\mathsf{half}(y)) = y$ w.r.t. $\mathcal{C}_{\mathsf{half}}$, for example, this conjecture has the desired property

$$AX_{\mathcal{T}_C} \cup \mathcal{R} \models_{\text{ind}} \mathsf{dbl}(\mathsf{half}(\mathsf{s}(\mathsf{s}(x)))) = \mathsf{s}(\mathsf{s}(x)) \;\Rightarrow\; \mathsf{dbl}(\mathsf{half}(x)) = x. \tag{7}$$

For many equations $r_1 = r_2$, one does not only have $r_1[s_i^*] \to_{\mathcal{R}}^* C_i[r_1[t_{i,1}^*], \ldots, r_1[t_{i,n_i}^*]]$ for all $\langle s_i^*, \{t_{i,1}^*, \ldots, t_{i,n_i}^*\}\rangle \in \mathcal{C}$, but also $r_2[s_i^*] = D_i[r_2[t_{i,1}^*], \ldots, r_2[t_{i,n_i}^*]]$ for some constructor ground contexts C_i and D_i. Property $(*)$ is ensured if whenever $\theta(r_1)$ is not equivalent to $\theta(r_2)$ for some instantiation, then $\sigma(r_1)$ is also not equivalent to $\sigma(r_2)$ for every substitution σ that is greater than θ w.r.t. the induction relation induced by the cover set. A sufficient requirement for this is that the contexts C_i added around r_1 are always at most as big as the contexts D_i added around r_2. This is made precise below.

Definition 10 *Given a monotonic ordering $\prec$ on constructor terms which is stable under substitutions, an equation $r_1 = r_2$* maintains $\prec$ *under a cover set $\mathcal{C}$ w.r.t. $\mathcal{R}$ iff $C_i[x^*] \preceq D_i[x^*]$ for all $1 \le i \le m$, where C_i and D_i are constructor ground contexts in*

$$\begin{array}{lcl} r_1[s_i^*] & \to_{\mathcal{R}}^* & C_i[r_1[t_{i,1}^*], \ldots, r_1[t_{i,n_i}^*]] \;\; \textit{and} \\ r_2[s_i^*] & = & D_i[r_2[t_{i,1}^*], \ldots, r_2[t_{i,n_i}^*]]. \end{array}$$

The following lemma proves that for equations which maintain an ordering, each induction conclusion indeed implies its induction hypothesis.

Lemma 1 *Let $\mathcal{R}$, $\mathcal{C}$, $\prec$ be as in Def. 10 and let $r_1 = r_2$ maintain $\prec$ under $\mathcal{C}$ w.r.t. $\mathcal{R}$. Then for all $1 \le i \le m$ with $n_i = 1$, $AX_{\mathcal{T}_C} \cup \mathcal{R} \models_{\text{ind}} r_1[s_i^*] = r_2[s_i^*] \;\Rightarrow\; r_1[t_{i,1}^*] = r_2[t_{i,1}^*]$.*

Now we prove that if $r_1 = r_2$ maintains an ordering, then $c_{r_1=r_2}$ is indeed exact.

Theorem 7 *[11] Let $r_1 = r_2$ be a radical equation which maintains some ordering $\prec$ under $\mathcal{C}$ w.r.t. $\mathcal{R}$ and let $c_{r_1=r_2}$ be its correctness predicate whose definition is included in $\mathcal{R}$. Then $AX_{\mathcal{T}_C} \cup \mathcal{R} \models_{\text{ind}} r_1 = r_2 \;\Leftrightarrow\; c_{r_1=r_2}(y^*) = \mathsf{true}$.*[8]

[8] A more general version of this theorem can be proved in which a conjecture does not have to be radical, and further, it is not necessary for the induction scheme of a cover set to have at most one induction hypothesis in every subgoal.

In a proof attempt of $\mathsf{half}(y) = \mathsf{s}(0)$, we obtain $C_1 = 0$, $D_1 = \mathsf{s}(0)$ and $C_2 = 0$, $D_2 = \mathsf{s}(0)$. In the induction step case, the left-hand side $\mathsf{half}(\mathsf{s}(\mathsf{s}(x)))$ evaluates to $\mathsf{s}(\mathsf{half}(x))$, i.e., we have $C_3 = \mathsf{s}(\Box)$, whereas $D_3 = \Box$. There does not exist an ordering $\prec$ such that $C_i[x^*] \preceq D_i[x^*]$ for all i, since $C_1 \preceq D_1$ would imply $0 \prec \mathsf{s}(0)$ and $C_3[0] \preceq D_3[0]$ would imply $\mathsf{s}(0) \prec 0$ which contradicts the transitivity and irreflexivity of $\prec$. Thus, $\mathsf{half}(y) = \mathsf{s}(0)$ does not maintain any ordering under $\mathcal{C}_{\mathsf{half}}$ and its correctness predicate is not exact.

The above analysis of exactness of correctness predicates can also be useful for fixing faulty conjectures, an objective for which correctness predicates were introduced by Protzen [26]. Since an exact correctness predicate precisely characterizes all instantiations on which the faulty conjecture is true, it can be used to fix the faulty conjecture into the "strongest theorem" possible.

5.3. Decidability of Boolean Combination of $\mathcal{C}$-provable Equations

Below, we require that all equations $r_1 = r_2$ occurring in a conjecture φ are radical and maintain some ordering under the same cover set $\mathcal{C}$.[9] By Thm. 7 , their correctness predicates $c_{r_1=r_2}$ are sound and exact. For example, $\mathsf{half}(y) = 0$ is radical and maintains the subterm ordering under $\mathcal{C}_{\mathsf{half}}$. We obtain the correctness predicate

$$c_{\mathsf{half}(y)=0}(0) \to \mathsf{true}, \quad c_{\mathsf{half}(y)=0}(\mathsf{s}(0)) \to \mathsf{true}, \quad c_{\mathsf{half}(y)=0}(\mathsf{s}(\mathsf{s}(x))) \to \mathsf{false}.$$

Given a correctness predicate c_φ, $c_{\neg\varphi}$ is generated by replacing the result true by false and the result false by true whereas right-hand sides of the form $c_\varphi(t^*)$ are replaced by $c_{\neg\varphi}(t^*)$. In the above example this yields

$$c_{\neg\mathsf{half}(y)=0}(0) \to \mathsf{false}, \quad c_{\neg\mathsf{half}(y)=0}(\mathsf{s}(0)) \to \mathsf{false}, \quad c_{\neg\mathsf{half}(y)=0}(\mathsf{s}(\mathsf{s}(x))) \to \mathsf{true}.$$

This correctness predicate is sound and exact for the conjecture $\neg\mathsf{half}(y) = 0$.

A straightforward construction for $c_{\varphi_1\wedge\varphi_2}$ from c_{φ_1} and c_{φ_2}, however, does not work if one rule $c_{\varphi_1}(s^*) \to c_{\varphi_1}(t^*)$ leads to a recursive call, but the other has the form $c_{\varphi_2}(s^*) \to \mathsf{true}$. If we make $c_{\varphi_1\wedge\varphi_2}(s^*) \to c_{\varphi_1\wedge\varphi_2}(t^*)$, then we may lose the exactness of the correctness predicate, since it could be that $c_{\varphi_2}(t^*) \to^* \mathsf{false}$. This is evident if one tries to construct the correctness predicate for φ : $\mathsf{dbl}(\mathsf{half}(y)) = y \wedge \neg\mathsf{half}(y) = 0$.

$$c_\varphi(0) \to \mathsf{false}, \quad c_\varphi(\mathsf{s}(0)) \to \mathsf{false}, \quad c_\varphi(\mathsf{s}(\mathsf{s}(x))) \to c_\varphi(x).$$

This correctness predicate is not exact, since it is always false, whereas φ is true for all even numbers greater than 0. Even worse, the resulting correctness predicate for the negated conjecture $\neg\varphi$ would not even be sound (since it would always be true whereas $\neg\varphi$ is false for all even numbers greater than 0).

To avoid this problem, *basic* correctness predicates (denoted $b_{r_1=r_2}$) are defined in [11], where for recursive pairs $\langle s^*, \{t^*\}\rangle \in \mathcal{C}$, it is required that we have the rule $b_{r_1=r_2}(s^*) \to b_{r_1=r_2}(t^*)$, but not the rule $b_{r_1=r_2}(s^*) \to \mathsf{false}$. For a radical equation maintaining an ordering with respect to a cover set $\mathcal{C}$, under certain conditions on $\mathcal{C}$ one can always obtain such a basic correctness predicate. To this end, $\mathcal{C}$ is *extended* so that the equation remain radical and maintains an ordering with respect to the extended cover set; see [11] for details.

Definition 11 *Let $\mathcal{R}$ be a convergent sufficiently complete rewrite system and let $\mathcal{C}$ be a complete well-founded cover set such that for all $\langle s^*, \{t_1^*, \ldots, t_n^*\}\rangle \in \mathcal{C}$, $0 \leq n \leq 1$, and for two different pairs $\langle s^*, \{t^*\}\rangle$, $\langle s^{*\prime}, \{t^{*\prime}\}\rangle \in \mathcal{C}$, there does not exist a substitution μ with $t^*\mu = s^{*\prime}\nu\mu$ for a variable renaming ν. Let φ be a quantifier-free formula such that all equations in φ are radical and maintain some ordering under $\mathcal{C}$ w.r.t. $\mathcal{R}$.*

[9] Different equations in a conjecture may have to be proved using different cover sets; these cover sets can often be combined into a single cover set to generate a single induction scheme using merging (cf. Section 3.4). Further, it is not necessary for different equations to maintain the same monotonic ordering.

Let $\mathcal{C}' = \{\langle s_1^*, \{t_{1,1}^*, \ldots, t_{1,n_1}^*\}\rangle, \ldots, \langle s_m^*, \{t_{m,1}^*, \ldots, t_{m,n_m}^*\}\rangle\}$ be the extension of $\mathcal{C}$, let $r_1[s_i^*] \to_{\mathcal{R}}^* C_i[r_1[t_{i,1}^*], \ldots, r_1[t_{i,n_i}^*]]$ for a constructor ground context C_i. Then the basic correctness predicate b_φ under $\mathcal{C}$ is defined by the following rules (analogous rules are used for formulas containing $\vee$, $\Rightarrow$, $\Leftrightarrow$):

$$b_{r_1=r_2}(s_i^*) \to \begin{cases} \mathsf{true}, & \text{if } C_i = r_2[s_i^*] \text{ is valid in } \mathcal{T}_C \text{ and } n_i = 0, \\ \mathsf{false}, & \text{if } \neg C_i = r_2[s_i^*] \text{ is valid in } \mathcal{T}_C \text{ and } n_i = 0, \\ b_{r_1=r_2}(t_{i,1}^*), & \text{if } n_i = 1, \end{cases}$$

$$b_{\neg\varphi'}(s_i^*) \to \begin{cases} \mathsf{true}, & \text{if we have the rule } b_{\varphi'}(s_i^*) \to \mathsf{false}, \\ \mathsf{false}, & \text{if we have the rule } b_{\varphi'}(s_i^*) \to \mathsf{true}, \\ b_{\neg\varphi'}(t_{i,1}^*), & \text{if we have the rule } b_{\varphi'}(s_i^*) \to b_{\varphi'}(t_i^*), \end{cases}$$

$$b_{\varphi_1\wedge\varphi_2}(s_i^*) \to \begin{cases} \mathsf{true}, & \text{if we have the rules } b_{\varphi_1}(s_i^*) \to \mathsf{true} \text{ and } b_{\varphi_2}(s_i^*) \to \mathsf{true}, \\ \mathsf{false}, & \text{if we have the rules } b_{\varphi_1}(s_i^*) \to \mathsf{false} \text{ or } b_{\varphi_2}(s_i^*) \to \mathsf{false}, \\ b_{\varphi_1\wedge\varphi_2}(t_{i,1}^*), & \text{if we have the rules } b_{\varphi_1}(s_i^*) \to b_{\varphi_1}(t_{i,1}^*) \text{ and } b_{\varphi_2}(s_i^*) \to b_{\varphi_2}(t_{i,1}^*). \end{cases}$$

In our example, we obtain

$$\begin{array}{rcl@{\qquad}rcl} b_{\mathsf{half}(y)=0}(0) & \to & \mathsf{true}, & b_{\mathsf{dbl}(\mathsf{half}(y))=y}(0) & \to & \mathsf{true}, \\ b_{\mathsf{half}(y)=0}(\mathsf{s}(0)) & \to & \mathsf{true}, & b_{\mathsf{dbl}(\mathsf{half}(y))=y}(\mathsf{s}(0)) & \to & \mathsf{false}, \\ b_{\mathsf{half}(y)=0}(\mathsf{s}^2(0)) & \to & \mathsf{false}, & b_{\mathsf{dbl}(\mathsf{half}(y))=y}(\mathsf{s}^2(0)) & \to & \mathsf{true}, \\ b_{\mathsf{half}(y)=0}(\mathsf{s}^3(0)) & \to & \mathsf{false}, & b_{\mathsf{dbl}(\mathsf{half}(y))=y}(\mathsf{s}^3(0)) & \to & \mathsf{false}, \\ b_{\mathsf{half}(y)=0}(\mathsf{s}^4(x)) & \to & b_{\mathsf{half}(y)=0}(\mathsf{s}^2(x)). & b_{\mathsf{dbl}(\mathsf{half}(y))=y}(\mathsf{s}^4(x)) & \to & b_{\mathsf{dbl}(\mathsf{half}(y))=y}(\mathsf{s}^2(x)). \end{array}$$

For $\mathsf{dbl}(\mathsf{half}(y)) = y \wedge \neg\mathsf{half}(y) = 0$,

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl} b_\varphi(0) & \to & \mathsf{false}, & b_\varphi(\mathsf{s}(0)) & \to & \mathsf{false}, & b_\varphi(\mathsf{s}^2(0)) & \to & \mathsf{true}, \\ b_\varphi(\mathsf{s}^3(0)) & \to & \mathsf{false}, & b_\varphi(\mathsf{s}^4(x)) & \to & b_\varphi(\mathsf{s}^2(x)). & & & \end{array}$$

Theorem 8 (Decidability of Arbitrary Conjectures) *Let $\mathcal{R}$, $\mathcal{C}$, φ be as in Def. 11. Then inductive validity of φ is decidable (by checking whether all non-recursive rules of b_φ have the right-hand side* true*, where b_φ is the basic correctness predicate for φ under $\mathcal{C}$).*

The condition of an equation being radical and maintaining an orderings can be checked automatically (by using orderings from the area of term rewrite systems which are amenable to automation). To decide inductive validity of φ, construct the rules for the basic correctness predicate b_φ (which can be done automatically) and check whether there is no rule of the form $b_\varphi(\ldots) \to \mathsf{false}$.

As another example, the equations $\mathsf{dbl}(y) = y$ and $y = 0$ are both radical. The equation $\mathsf{dbl}(y) = y$ maintains the superterm ordering and $y = 0$ maintains the subterm ordering. So inductive validity of the conjecture $\mathsf{dbl}(y) = y \Rightarrow y = 0$ is decidable. The decision procedure constructs the following rules for the basic correctness predicate.

$$\begin{array}{rcl@{\qquad}rcl} b_{\mathsf{dbl}(y)=y\Rightarrow y=0}(0) & \to & \mathsf{true}, & b_{\mathsf{dbl}(y)=y\Rightarrow y=0}(\mathsf{s}(0)) & \to & \mathsf{true}, \\ b_{\mathsf{dbl}(y)=y\Rightarrow y=0}(\mathsf{s}(\mathsf{s}(x))) & \to & b_{\mathsf{dbl}(x)=x}(\mathsf{s}(x)). & & & \end{array}$$

The conjecture in our example is valid since the above basic correctness predicate is always true. We can thus *decide* conjectures which were up to now hard problems for inductive theorem provers. In fact, virtually all existing inductive provers fail in verifying $\mathsf{dbl}(y) = y \Rightarrow y = 0$.[10] The reason is that the induction conclusion $\mathsf{dbl}(\mathsf{s}(x)) = \mathsf{s}(x) \Rightarrow \mathsf{s}(x) = 0$ can be evaluated to $\neg\mathsf{s}(\mathsf{dbl}(x)) = x$, but there is no way to apply the induction hypothesis $\mathsf{dbl}(x) = x \Rightarrow x = 0$ and thus, the proof of the induction step case does not succeed.

[10] This problem was pointed out to us by U. Kühler.

6. Generating Lemmas using Decision Procedures

A major challenge in attempting proofs by induction is to determine intermediate lemmas needed to complete a proof. In this section, we discuss two topics in which a decision procedure of a data structure plays an important role in generation and speculation of intermediate lemmas.

In [23], Kapur and Subramaniam proposed an approach for automatically generating lemmas of the form $h(\ldots g(\ldots h(\ldots)\ldots)\ldots) = r$ from $\mathcal{T}$-based recursive definitions of h, g, f using a decision procedure for $\mathcal{T}$; r is a $\mathcal{T}$-term. Apart from the possible use of these lemmas in proving other nontrivial properties about functions appearing in them, there are at least two additional types of benefits of generating such lemmas. If a user is surprised to see a particular lemma following from a definition, i.e., he/she did not expect the property to be true, this can provide useful insight about a bug in the definition. Otherwise, it enhances user's confidence in the formulation of the definition. A decision procedure for $\mathcal{T}$ is used to derive the right-hand side of such a lemma once its left-hand side can be hypothesized.

The procedure introduced in [23] can be run in the background after the $\mathcal{T}$-based recursive definitions are formulated; lemmas are generated without any help from the user. In the next subsection, we review the main ideas of the approach and give an example. Details, including the precise description of the procedure, can be found in [23].

The second topic of discussion is the generalization heuristic employed in most theorem provers. Our theorem prover *RRL*, for instance, employs an extremely aggressive approach in the course of a proof attempt for speculating intermediate conjectures which works well with the backtracking facility. Before a proof of a conjecture is attempted, common nonvariable subterms in the conjecture are identified for possible generalization by fresh variables (under certain conditions); various generalizations of the conjecture are attempted for possible proofs before trying the original conjecture. Contrary to the experience reported in [4, 6] our experience has been quite positive; if a counter-example to a proposed generalization is identified or a proof attempt of a proposed generalization does not appear to be making progress, backtracking is performed.

In case function definitions are given on a data structure that has a decision procedure associated with its theory, e.g., Presburger arithmetic, then merely generalizing identical nonvariable subterms by distinct variables often leads to unsound generalizations. As shown in [17], a decision procedure can be used to check for equivalence of subterms. Equivalent subterms are abstracted by a variable. Determining whether a term appears as a subterm within a given term is also done using the decision procedure. These ideas are briefly reviewed in the second subsection; for details, the reader should refer to [17].

6.1. Generating Simple Lemmas from Definitions

The key idea of the method proposed in [23] is to identify terms of the form $f(s_1, \ldots, s_k)$, where each s_i is a $\mathcal{T}$-term, and hypothesize a simple conjecture $f(s_1, \ldots, s_k) = r$, where r is a *parametric* general term in $\mathcal{T}$ expressed using all the variables appearing in the left-hand side and some parameters whose values have to be determined. Depending upon $\mathcal{T}$, r may take many different forms, depending upon the number of canonical forms of $\mathcal{T}$-terms. Using the definition of f, constraints on parameters in r are collected and these constraints are checked for consistency; if found consistent, then for every assignment δ of values of parameters that satisfy these constraints, a simple lemma $f(s_1, \ldots, s_k) = \delta(r)$ is generated. If there are infinitely many such assignments satisfying the constraints, then a finite description of these assignments is sought to generate finitely many lemmas, as otherwise, the procedure will diverge generating infinitely many lemmas. If constraints are consistent only for certain instances of $f(s_1, \ldots, s_k)$, then those instances can be generated as well. If no instance of $f(s_1, \ldots, s_k)$ can be identified for which constraints on parameters are consistent, then no lemma with an instance of $f(s_1, \ldots, s_k)$ is generated. In order to be able to generate most general lemmas, various instances of $f(s_1, \ldots, s_k)$ leading to consistent constraints on parameters can be combined and abstracted. We first illustrate these ideas using an example; this is followed by a detailed discussion of the key steps. The general procedure with additional examples is given in [23].

Consider e2plus below which is defined on PA-terms; the functione2plus(x, y) computes $2^x + y$. Here,

$\mathsf{s}(t)$ is a shorthand for $t+1$. We are interested in generating lemmas about e2plus such that their right-hand sides are expressible in PA.

1. $\mathsf{e2plus}(0,0) \to 1$,
2. $\mathsf{e2plus}(0,\mathsf{s}(y)) \to \mathsf{s}(\mathsf{e2plus}(0,y))$,
3. $\mathsf{e2plus}(\mathsf{s}(x),0) \to \mathsf{e2plus}(x,0) + \mathsf{e2plus}(x,0)$,
4. $\mathsf{e2plus}(\mathsf{s}(x),\mathsf{s}(y)) \to \mathsf{s}(\mathsf{e2plus}(\mathsf{s}(x),y))$.

Begin with the conjecture $\mathsf{e2plus}(x,y) = k_1\ x + k_2\ y + k_3$ with the goal of finding k_1, k_2, k_3.[11] Constraints are generated from the rules. Rule 1 gives $s1 : k_3 = 1$. The second constraint is generated from rule 2 by replacing the left-hand side and the recursive call with $k_1\ 0 + k_2\ \mathsf{s}(y) + k_3$ and $k_1\ 0 + k_2\ y + k_3$. It simplifies to $s2 : k_2 = 1$. The third and fourth constraints are generated in a similar fashion from rules 3 and 4, and they simplify to $s3 : k_1 = k_1\ x + k_3$ and $s4 : k_2 = 1$. These constraints are not satisfiable for all values of x and y. So, there is no solution for all values of x, and hence e2plus cannot be expressed in PA.

As the reader would notice, there are two kinds of constraints: (i) constraints purely in terms of parameters (e.g., $s1, s2, s4$), called *parametric* constraints, and (ii) constraints involving both parameters and variables (e.g. $s3$). Some of the second type of constraints can be transformed to parametric constraints (hence true for all values of variables) (for $s3$, this would mean $k_1 = 0, k_3 = 0$); but they are also true for specific values of variables and parameters (for $s3$, this could mean that for $x = 1$, $k_3 = 0$ and k_1 can take any value). Such a constraint is called *strongly satisfiable*. However, there are also constraints involving both parameters and variables which cannot be transformed to parametric constraints (and thus they do not hold for all values of variables) and which can hold for only some values of variables and parameters; they are called *weakly satisfiable* (e.g., the constraint $(k_1 + k_2)x_1 + k_2x_2 + k_1 - 1 = 0$ is such an example).

Now conjectures with proper instances of $\mathsf{e2plus}(x,y)$ as their left-hand sides can be speculated. In order to attempt to generate most general conjectures, maximal consistent subsets of the set of inconsistent constraints $\{s1, s2, s3, s4\}$ are identified. One such maximal consistent subset of the above inconsistent set of constraints is $\{s1, s2, s4\}$ giving $k_2 = 1, k_3 = 1$ with k_1 being unconstrained. For these values of parameters, constraint $s3$ can be solved for specific values of variables. Particularly, if x = 0, then $s3$ reduces to $s3.1$: $k_1 = k_3$, which can be consistently added to $\{s1, s2, s4\}$[12]. The rules associated with the instances are computed by splitting the rule using the variable substitution, e.g:, rule 3 is split by $\{x \to 0\}$ into two rules; how this splitting is done is explained in [23].

3.1 $\mathsf{e2plus}(\mathsf{s}(0),0) \to 2$,
3.2 $\mathsf{e2plus}(\mathsf{s}(\mathsf{s}(x)),0) \to \mathsf{e2plus}(x,0) + \mathsf{e2plus}(x,0) + \mathsf{e2plus}(x,0) + \mathsf{e2plus}(x,0)$.

The constraint for rule 3.1 is $s3.1 : k_1 + k_3 = 2$ and the constraint for rule 3.2 is $s3.2 : 2k_1 = 3k_1\ x + 3k_3$. The rules to be considered for speculating a conjecture are $\{1, 2, 3.1, 4\}$ based on a maximal consistent subset $\{s1, s2, s3.1, s4\}$ of constraints. ¿From the left-hand side of these rules, a possible conjecture with $\mathsf{e2plus}(0,y)$ as the left-hand side is generated (since the left-hand sides of rules 1 and 2 are $\mathsf{e2plus}(0,0), \mathsf{e2plus}(0,\mathsf{s}(y))$).

The rules $\{1, 2, 3.1, 3.2, 4\}$ are analyzed to pick their instances which are likely to be used to compute $\mathsf{e2plus}(0,y)$. Rules 1 and 2 constitute such a complete set since any ground instance of $\mathsf{e2plus}(0,y)$ can be normalized to a PA-term using them. The constraints generated from rules 1 and 2 have the solution $\{k_2 = 1, k_3 = 1\}$, which generates the lemma $\mathsf{e2plus}(0,y) = \mathsf{s}(y)$.

Rules associated with the maximal set $\{1, 2, 3.1, 4\}$ may also be expanded to generate additional rules for generating left-hand sides for conjectures. The expansion is done by choosing a rule whose right-hand side is a PA-term, e.g: rule 3.1, and splitting by unifying the recursive calls in other rules with the left-hand side of rule 3.1. The unification of the recursive call $\mathsf{e2plus}(\mathsf{s}(x),y)$ in rule 4 with $\mathsf{e2plus}(\mathsf{s}(0),0)$ of rule 3.1 produces four equivalent rules two of which are:

4.1 $\mathsf{e2plus}(\mathsf{s}(0),\mathsf{s}(0)) \to 3$, 4.2 $\mathsf{e2plus}(\mathsf{s}(0),\mathsf{s}(\mathsf{s}(y))) \to \mathsf{s}(\mathsf{s}(\mathsf{e2plus}(\mathsf{s}(0),y)))$.

[11] This corresponds to the no-theory condition for e2plus, which is decidable for the theory of free constructors $\mathcal{C}$ as well as for $\mathcal{T}_{PA}$.

[12] Other values of x produce instances with k_3 = 0, which contradicts $s1$ and hence cannot be added to $\{s1, s2, s4\}$.

The conjecture with the left-hand side $\mathsf{e2plus}(\mathsf{s}(0), y)$ can now be hypothesized by abstracting from the left-hand sides of rules 3.1, 4.1, and 4.2, which completely define $\mathsf{e2plus}(\mathsf{s}(0), y)$. The associated constraints, $s3.1 : k_1 + k_3 = 2$, $s4.1 : k_1 + k_2 + k_3 = 3$ and $s4.2 : k_2 = 1$, are solved and give $\mathsf{e2plus}(\mathsf{s}(0), y) \rightarrow y + 2$ as the new lemma.

6.2. Key Steps of the Approach

There are three major steps in the procedure for generating lemmas. The discussion below uses PA. For details, the reader should refer to [23].

6.2.1. Generating Right-Hand Side of a Conjecture

Given a conjecture of the form $f(s_1, \ldots, s_k) = r$, where r is a parametric term over PA, find values δ of parameters in r, if any, such that $f(s_1, \ldots, s_k) = \delta(r)$ is an inductive theorem.

Let $lhs = f(s_1, \ldots, s_k), r = \Sigma k_i\ x_i + k_0$. Also assume that the set of rules in the definition of f needed to normalize ground instances of lhs is already computed, denoted as D_{lhs}. For each rule $l_i \rightarrow r_i$ in D_{lhs}, a parametric equation is generated by replacing each term of the form $f(t_1, \ldots, t_k)$ in the rule by $\sigma(\Sigma k_i x_i + k_0)$, where $\sigma(f(s_1, \ldots, s_k)) = f(t_1, \ldots, t_k)$. To ensure that a parametric equations can be generated corresponding to a rule $l_i \rightarrow r_i$, l_i must match with lhs and each recursive call t to f in r_i must either match with lhs or $\sigma(t)$ must simplify to a $\mathcal{T}$-term.

Let S be the finite set of all the constraints so generated. A constraint may be simplified using $\mathcal{T}_{PA}$ to a parametric constraint. For example, given two parametric expressions, say $k_1\ x_1 + k_2\ x_2 + k_2\ x_2 + k_0 = k_3\ x_1 + k_3\ x_1 + k_2\ x_2 + k_4\ x_2$, where k_0, k_1, k_2, k_3, k_4 are parameters, the most general solution is $k_1 = k_3 + k_3, k_4 = k_2, k_0 = 0$. If S includes an unsatisfiable constraint or a weakly satisfiable parametric equation (in which both parameters as well as variables appear), then S is not consistent. Values of parameters constrained by different parametric constraints may clash as well.

If S is consistent implying that for all values of variables, an assignment of parameter values can be found satisfying every constraint in S, generate a finite representation of all such assignments of parameters. (This problem is related to the unification problem over $\mathcal{T}$.) For every most general solution δ of the set of parametric constraints generated from S, the conjecture $lhs = \delta(\Sigma k_i x_i + k_0)$, is a lemma. And, this also implies that lhs is expressible in PA. If an assignment of parameters cannot be found for which each constraint in S is true for all values of variables, then generate a finite representation of values of variables and the corresponding assignment of parameters for which the constraints in S are true. These assignments then give instances of $lhs = r$ as lemmas.

6.2.2. Generating Maximal Consistent Subsets

If the set S of parametric equations generated from $lhs = f(s_1, \ldots, s_k)$ is inconsistent, maximal consistent subsets of S can be used to generate instances of lhs possibly serving as the left-hand sides of new conjectures about f.

First, every unsatisfiable parametric equation is deleted from S. The remaining equations are partitioned into the subsets S_{cn}, S_{st}, S_{wk}, standing for parametric constraints, strongly satisfiable parametric equations (which involve both variables and parameters) and weakly satisfiable (also called weak) parametric equations (which also involve both variables and parameters). A strongly satisfiable parametric equation $st : \Sigma p_i x_i + p_0 = 0$ in S_{st} can either be viewed as a conjunction of parametric constraints $c : p_0 = 0, \ldots, p_m = 0$ which can be included in S_{cn}, or it can be viewed as a weakly satisfiable parametric equation and be included in S_{wk}. For each $st \in S_{st}$, there are two cases leading to exponentially many possibilities, each consisting of the set of parametric constraints (purely in terms of parameters) and the set of weak parametric equations both in parameters and variables. For each such set of constraints and any maximal consistent subset of parametric constraints, a most general solution δ for parameters is computed; δ is then applied on each weak parametric equation involving both parameters and variables to compute

the values of variables, if any, such that the instantiated equation is consistent. For any such substitution of variables of the weak parametric equations, the corresponding rule is split using the variable instantiations. A rule set from D_{lhs} is then identified corresponding to the maximal consistent subset of constraints. This rule set is then used to speculate possible left-hand sides of conjectures by abstracting from its left-hand sides.

For example, consider the inconsistent set of parametric equations $S = \{s1 : k_3 = 1,\ s2 : k_2 = 1,\ s3 : k_1 = k_1\ x + k_3,\ s4 : k_2 = 1\}$ generated from D_{e2plus}. $S_{cn} = \{s1 : k_3 = 1;\ s2 : k_2 = 1;\ s4 : k_2 = 1\}$, $S_{st} = \{k_1\ x + k_3 = k_1\}$, and $S_{wk} = \varnothing$. There are two possibilities based on $\{k_1\ x + k_3 = k_1\}$: $\{\langle\{s1 : k_3 = 1;\ s2 : k_2 = 1;\ s3 : k_1 = 0, k_3 = 0;\ s4 : k_2 = 1\}, \varnothing\rangle, \langle\{s1 : k_3 = 1;\ s2 : k_2 = 1;\ s4 : k_2 = 1\}, \{s3 : k_1\ x + k_3 = k_1\}\rangle\}$. There are two maximal consistent subsets $MCS_1 = \{k_1 = 0, k_3 = 0, k_2 = 1\}$ and $MCS_2 = \{k_3 = 1, k_2 = 1\}$. Corresponding to MCS_1, the rule set is $\{2, 3, 4\}$. For MCS_2, an instance of rule 3 generated from $s3$ is computed requiring x to be 0; the resulting rule set is $\{1, 2, 3.1, 4\}$.

6.2.3. Identifying Left-Hand Sides and their Complete Definitions

Given a subset R of rules of D_f which possibly generate a consistent set of parametric equations, possible left-hand sides of conjectures need to be speculated. For every speculated left-hand side $lhs = f(s_1, \ldots, s_k)$ of a conjecture, it must be ensured that every ground instance of lhs can be normalized to a ground $\mathcal{T}$-term using the rules in R; if not, additional rules from D_f are added to R to ensure this property (hopefully without violating the consistency condition).

Rules in R are first refined by splitting based on unifying recursive calls with the left-hand sides of nonrecursive rules in R. Let $l_i \to r_i$ be a non-recursive rule in R (i.e., r_i is a PA-term). For every recursive rule $l_j \to r_j$ in R, replace it by $split(\{l_j \to r_j\}, \sigma)$ if a recursive call in r_j unifies with l_i with the most general unifier σ. The expansion is performed for all possible pairs of non-recursive rules $l_i \to r_i$ and recursive rules $l_j \to r_j$ in R. Splitting of a rule is guided by the substitution of variables and is precisely defined in [23].

Once a complete set of rules for computing every instance of lhs has been identified, the left-hand sides of these rules are abstracted to generate k-tuples of terms to construct the left-hand sides of new conjectures to be speculated (see the procedure ABS in [23] for details). For each such tuple in the output, $lhs = f(s_1, \ldots, s_k)$, its complete definition D_{lhs} as a set of rules is computed from R (see the procedure $complete$ in [23]). Then the solutions for the constraints resulting from D_{lhs} determine the right-hand sides of the new lemmas $lhs = r$. These steps are repeated until it is not possible to proceed any further or enough lemmas have been generated (since we suspect that the procedure can, in general, generate infinitely many lemmas).

6.3. Generating Lemmas with Multiple Defined Function Symbols

The above discussion was limited to generating lemmas of the form $f(s_1, \ldots, s_k) = r$, where r and each s_i are $\mathcal{T}$-terms. Using function symbols with compatible $\mathcal{T}$-based definitions, lemmas whose left-hand sides include multiple occurrences of $\mathcal{T}$-based functions can also be generated. Many nontrivial lemmas needed in the verification of arithmetic circuits [18, 20, 21] are of this form. Given PA-based function symbols g and f such that g is compatible with f, a conjecture $g(y_1, \ldots, f(x_1, \ldots, x_n), \ldots, y_k) = \Sigma k_i x_i + \Sigma k_j y_j + k_0$ can be speculated where x_i's and y_j's are distinct variables and f appears only in the inductive argument positions of g (on which the definition of g recurses). The procedure is the same as in the previous subsection. This is illustrated using an example below.

Refer to the definitions of function symbols half and mod2 given in Section 4.2. The procedure in the previous subsection will not generate any simple lemma with an instance of $\mathsf{mod2}(x)$ or $\mathsf{half}(x)$. Compatibility sequences of function symbols can be generated easily.[13] For instance, half is compatible with mod2.

[13] Note that besides the rules in the definition of f and g, other applicable rules may be used to simplify terms to check compatibility of functions f and g. Properties of function symbols such as associativity and commutativity (*AC*) may also be used.

Given $\mathsf{half}(\mathsf{mod2}(x)) = k_1\ x + k_2$, where parameters k_1, k_2 are unknown, parametric equations are generated using the rules defining mod2. From rule 1, x is instantiated to 0, giving $\mathsf{half}(\mathsf{mod2}(0)) = k_1\ 0 + k_2$, which simplifies to the constraint $k_2 = 0$. From rule 2, x is instantiated to 1, giving $\mathsf{half}(\mathsf{mod2}(1)) = k_1\ 1 + k_2$, which simplifies to the constraint $k_1 + k_2 = 0$. From rule 3, x is instantiated to $\mathsf{s}(\mathsf{s}(u))$ giving the conclusion subgoal $\mathsf{half}(\mathsf{mod2}(\mathsf{s}(\mathsf{s}(u)))) = k_1\ u + k_1 + k_1 + k_2$ assuming the induction hypothesis $\mathsf{half}(\mathsf{mod2}(u)) = k_1\ u + k_2$, obtained by instantiating x to be u. After simplifying and using the induction hypothesis, we get $k_1\ u + k_2 = k_1\ u + k_1 + k_1 + k_2$, which simplifies to $k_1 + k_1 = 0$. These three constraints are consistent, giving $k_1 = k_2 = 0$ as the solution. Thus, the lemma $\mathsf{half}(\mathsf{mod2}(x)) = 0$ is generated.

If a conjecture generated from a compatibility sequence of function definitions cannot be expressed in $\mathcal{T}$, then appropriate instances of the conjecture are computed exactly in the same way as discussed in the previous subsection. Some of the lemmas generated by the proposed approach based on the theories of PA and finite lists are given below; more details can be found in [23].

$$\begin{aligned}
\mathsf{bton}(\mathsf{leftshift}(\mathsf{ntob}(x))) &= x + x, & \mathsf{bton}(\mathsf{pad0}(\mathsf{ntob}(x))) &= x,\\
\mathsf{compl}(\mathsf{compl}(x)) &= x, & \mathsf{half}(\mathsf{mod2}(x)) &= 0,\\
\mathsf{mod2}(\mathsf{mod2}(x)) &= 0, & & \\
\mathsf{ack}(1, x) &= x + 2, & \mathsf{ack}(2, x) &= x + x + 3,\\
\mathsf{rev}(\mathsf{rev}(x)) &= x, & \mathsf{app}(x, \mathsf{nil}) &= x,\\
\mathsf{bton}(\mathsf{ripplecarry}(\mathsf{ntob}(x), \mathsf{ntob}(x), \mathsf{ntob}(x))) &= x + x + x,\\
\mathsf{bton}(\mathsf{carrysave}(\mathsf{ntob}(x), \mathsf{ntob}(x), \mathsf{ntob}(x))) &= x + x + x.
\end{aligned}$$

Recall that bton and ntob convert a bit vector to a number and vice versa, and pad0 pads a bit vector; compl complements a bit vector; leftshift shifts a bit vector by one position. The functions ripplecarry and carrysave take three bit vectors and generate a bit vector, implementing the ripple carry and carry save adders, respectively. Definitions of these functions are given in [20]. The function rev reverses a list. The function ack computes the Ackermann function.

6.4. Enhancing Generalization Heuristic: Using Decision Procedure for Identifying Subterms

Most theorem provers use a simple heuristic of replacing common nonvariable subterms by variables to generalize a conjecture. Semantic analysis can be useful in the implementation of the generalization heuristic since a subterm may have multiple occurrences semantically but syntactically these multiple occurrences may not be apparent. Consider the following conjecture about gcd:

$$\mathsf{gcd}(x + y + 1, 2 \cdot x + 2 \cdot y + 2) = x + y + 1.$$

The first argument of gcd appears to be quite different from the second argument. If no semantic analysis is performed and syntactic equality is used to generalize common subterms, $\mathsf{gcd}(u, 2 \cdot x + 2 \cdot y + 2) = u$ is a generalized version of the conjecture by abstracting occurrences of $x + y + 1$ in the left-hand and right-hand side by a variable u. This generalization is not a valid formula since there are counterexamples. ¿From a semantic standpoint, however, the second argument of gcd is twice the first argument, which can be detected using $\mathcal{T}_{PA}$. Using this relationship between the first and second arguments, the generalized conjecture is $\mathsf{gcd}(u, 2 \cdot u) = u$, which can be proved.

Given a conjecture C of the form $l = r$ if $cond$, the heuristic identifies a maximal nonvariable subterm s occurring in at least two of l, r and $cond$. Instead of looking for syntactically identical subterms, look for semantically equivalent subterms. Since there may be many such maximal subterms, they are collected in a list as possible candidates for generalization. For every nonempty subset from this set of candidates, generate a generalized version C_g of the conjecture C by simultaneously replacing distinct subterms by distinct variables and try its proof. If any such generalized conjecture is proved, then the original conjecture is also proved. Otherwise, if all possible generalizations have been attempted unsuccessfully, then the original conjecture C is attempted.

For Presburger arithmetic, maximal subterms are identified as follows: for a given subterm s, if s is not a linear term, syntactic subterm check is performed; otherwise, if s is linear, other linear subterms are checked for occurrences of s using $\mathcal{T}_{PA}$ by querying whether there is a linear subterm $t \geq s$. If the answer is no, then s does not occur in t; otherwise, find the number of times s appears in t (this can be done by repeated query and subtraction from t until the result becomes smaller than s). Let $t = k * s + tr$, where k is a positive integer and tr is a linear term (smaller or noncomparable) than s. Then, s is abstracted as a new variable u with t replaced by $k * u + tr$. A complete description of the extended generalization procedure using $\mathcal{T}_{PA}$ with many examples is given in [17].

Using the results in [12], the generalization heuristic can be made a lot more powerful. Theorem 4 states conditions under which a nonvariable subterm that does not satisfy the no-theory condition can be safely replaced by a variable. This needs further investigation.

7. Concluding Remarks

The paper gives an overview of the cover set method for mechanizing inductive reasoning with a particular focus on how decision procedures for data structures can be exploited for semantic analysis. We have also discussed how induction can be integrated into decision procedures without compromising their automation. In particular, we have shown how a decision procedure can be used to

1. generate an induction scheme from an instance of a basic term $f(x_1, \ldots, x_k)$ in a given conjecture and the definition of f using the cover set method;
2. identify a large class of conjectures about defined functions based on syntactic conditions for which the cover set method serves as a decision procedure for their validity even though their proofs need to be done by induction;
3. generate lemmas from function definitions using decision procedures; and
4. enhance generalization heuristic using a decision procedure for speculating generalizations of conjectures based on identifying equivalent subterms.

We believe our approach allows an integration of inductive reasoning within fully automated tools like model checkers or compilers.

There are still many open problems which need to be investigated to move towards the goal of automatically deciding additional conjectures. The focus so far has been on 1-level proofs by induction, requiring that every subgoal in such a proof (after possible application of an induction hypothesis and simplification possibly using lemmas) either simplifies to a formula in a decidable theory or to a formula with defined symbols which can be safely generalized to a formula in a decidable theory. A typical induction proof however can lead to many subgoals which themselves may have to be proved by induction. The approach needs to be extended to handle such cases.

Since the proposed approach leads to a decision procedure for a class of quantifier-free formulas involving defined symbols, this gives an extended decidable theory with formulas in which defined symbols occur. Two types of bootstrapping is possible using the extended decidable theory. Firstly, new recursive definitions can be given using terms (containing defined symbols) in the extended decidable theory. In this way, a $\mathcal{T}$-based function definition can use other defined function symbols which themselves have $\mathcal{T}$-based definitions. Secondly, a subgoal in a proof attempt could simplify to a formula with defined symbols which is in the extended decidable theory.

It should also be possible to extend the conditions for safe generalizations in [12] beyond the theories of free constructors and of Presburger Arithmetic. Further, these ideas are of independent interest and can be useful for designing safe generalization heuristics for speculating intermediate lemmas. Moreover, there are lots of possibilities for generating lemmas from $\mathcal{T}$-based definitions in the background using quantifier-elimination methods for parametric formulas expressed in a decidable theory that admits quantifier-elimination.

Another promising topic for research is the failure analysis of induction schemes as proposed in Subramaniam's thesis [27] to identify a priori and weed out induction schemes which are not likely to succeed in a particular attempt.

Acknowledgement. This research was partially supported by an NSF ITR award CCR-0113611.

References

[1] Autexier, S., Hutter, D., Mantel, H., & Schairer, A. (1999) Inka 5.0 – A Logical Voyager. *Proc. CADE-16*, LNAI 1632.

[2] Baader, F. & Nipkow, T. (1998) *Term Rewriting and All That*. Cambridge University Press.

[3] Bouhoula, A. & Rusinowitch, M. Implicit Induction in Conditional Theories. *Journal of Automated Reasoning*, 14:189–235.

[4] Boyer, R. S. & Moore, J S. (1979) *A Computational Logic*, Academic Press.

[5] Boyer, R. S. & Moore, J S. (1988) *A Computational Logic Handbook.* Academic Press.

[6] Boyer, R. S. & Moore, J S. (1988) Integrating Decision Procedures Into Heuristic Theorem Provers: A Case Study of Linear Arithmetic. *Machine Intelligence*, 11:83–157.

[7] Bundy, A., Stevens, A., van Harmelen, F., Ireland, A. & Smaill, A. (1993) Rippling: A Heuristic for Guiding Inductive Proofs. *Artificial Intelligence*, 62:185-253.

[8] Dershowitz, N. (1987) Termination of Rewriting. *Journal of Symbolic Computation*, 3:69–116.

[9] Enderton, H. B. (1994) *A Mathematical Introduction to Logic*. 2nd edition, Harcourt/Academic Press.

[10] Franova, M. & Kodratoff, Y. (1992) Predicate Synthesis from Formal Specifications, *Proc. ECAI 92.*

[11] Giesl, J. & Kapur, D. (2001) Decidable Classes of Inductive Theorems. *Proc. IJCAR '01*, LNAI 2083, pp. 469–484.

[12] Giesl, J. & Kapur, D. (2003) Deciding Inductive Validity of Equations. *Proc. CADE'03*, LNAI 2741, pp. 17-31. An expanded version is Technical Report AIB-2003-03, 2003. Available from `http://aib.informatik.rwth-aachen.de`

[13] Kapur, D. (1994) An Automated Tool for Analyzing Completeness of Equational Specifications. *Proc. of International Symposium on Software Testing and Analysis (ISSTA)*, pp. 28–43.

[14] Kapur, D. (2001) Rewriting, Induction and Decision Procedures: A Case Study of Presburger Arithmetic. *Symbolic-Algebraic Methods and Verification Methods — Theory and Applications*, (eds. Alefeld, Rohn, Rump, and Yamamato), Springer Mathematics, Wien-NY, pp. 129–144.

[15] Kapur, D., Narendran, P., Rosenkrantz, D. & Zhang, H. (1991) Sufficient-Completeness, Quasi-Reducibility and Their Complexity. *Acta Informatica*, 28:311–350.

[16] Kapur, D. & Nie, X. (1994) Reasoning About Numbers in Tecton. *Proc. 8th International Symposium on Methodologies for Intelligent Systems, (ISMIS'94)*, pp. 57–70.

[17] Kapur, D. & Subramaniam, M. (1996) New Uses of Linear Arithmetic in Automated Theorem Proving by Induction. *Journal of Automated Reasoning*, 16:39–78.

[18] Kapur, D. & Subramaniam, M. (1996) Mechanically Verifying a Family of Multiplier Circuits. *Proc. Computer Aided Verification (CAV)*, LNCS 1102, pp. 135–146.

[19] Kapur, D. & Subramaniam, M. (1997) Mechanizing Verification of Arithmetic Circuits: SRT Division. *Proc. FSTTCS-17*, LNCS 1346, pp. 103–122.

[20] Kapur, D. & Subramaniam, M. (1998) Mechanical Verification of Adder Circuits using Powerlists. *Journal of Formal Methods in System Design*, 13(2):127–158.

[21] Kapur, D. & Subramaniam, M. (2000) Using an Induction Prover for Verifying Arithmetic Circuits. *International Journal of Software Tools for Technology Transfer*, 3(1):32–65.

[22] Kapur, D. & Subramaniam, M. (2000) Extending Decision Procedures with Induction Schemes. *Proc. CADE-17*, LNAI 1831, pp. 324–345.

[23] Kapur, D. & Subramaniam, M. (2003) Automatic Generation of Simple Lemmas from Recursive Definitions Using Decision Procedures–Preliminary Report. *Proc. ASIAN 2003*, LNCS.

[24] Kapur, D. & Zhang, H. (1995) An Overview of Rewrite Rule Laboratory (*RRL*). *Journal of Computer and Mathematics with Applications*, 29:91–114.

[25] Kaufmann, M., Manolios, P. & Moore, J S. (2000) *Computer-Aided Reasoning: An Approach.* Kluwer.

[26] Protzen, M. (1996) Patching Faulty Conjectures, *Proc. CADE-13*, LNAI 1104.

[27] Subramaniam, M. (1997) Failure Analyses in Inductive Theorem Provers. Ph.D. Thesis, Department of Computer Science, University of Albany, New York.

[28] Zhang, H, Kapur, D. & Krishnamoorthy, M. S. (1988) A Mechanizable Induction Principle for Equational Specifications. *Proc. CADE-9*, LNCS 310.

[29] Zhang, H. (1988). *Reduction, Superposition and Induction: Automated Reasoning in an Equational Logic.* Ph.D. Thesis, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY.

Deepak Kapur
Dept. of Computer Science
University of New Mexico
Albuquerque, NM 87131, USA
kapur@cs.unm.edu

Jürgen Giesl
LuFG Informatik II
RWTH Aachen
Ahornstr. 55
52074 Aachen, Germany
giesl@informatik.rwth-aachen.de

Mahadevan Subramaniam
Dept. of Computer Science
University of Nebraska
Omaha, NE, USA
msubramaniam@mail.unomaha.edu