Extending Decision Procedures with Induction Schemes

Deepak Kapur[1] ⋆ *and* **M. Subramaniam**[2]

[1] Department of Computer Science
University of New Mexico
Albuquerque, NM
kapur@cs.unm.edu
[2] HAL Computer Systems
Fujitsu Inc
Campbell, CA
subu@hal.com

Abstract. Families of function definitions and conjectures based in quantifier-free decidable theories are identified for which inductive validity of conjectures can be decided by the *cover set* method, a heuristic implemented in a rewrite-based induction theorem prover *Rewrite Rule Laboratory* (*RRL*) for mechanizing induction. Conditions characterizing definitions and conjectures are syntactic, and can be easily checked, thus making it possible to determine a priori whether a given conjecture can be decided. The concept of a $\mathcal{T}$*-based* function definition is introduced that consists of a finite set of terminating complete rewrite rules of the form $f(s_1, \cdots, s_m) \to r$, where $s_1, \cdots, s_m$ are interpreted terms from a decidable theory $\mathcal{T}$, and r is either an interpreted term or has non-nested recursive calls to f with all other function symbols from $\mathcal{T}$. Two kinds of conjectures are considered. *Simple* conjectures are of the form $f(x_1, \cdots x_m) = t$, where f is $\mathcal{T}$-based, x_i's are distinct variables, and t is interpreted in $\mathcal{T}$. *Complex* conjectures differ from simple conjectures in their left sides which may contain many function symbols whose definitions are $\mathcal{T}$-based and the nested order in which these function symbols appear in the left sides have the *compatibility property* with their definitions.

The main objective is to ensure that for each induction subgoal generated from a conjecture after selecting an induction scheme, the resulting formula can be simplified so that induction hypothesis(es), whenever needed, is applicable, and the result of this application is a formula in $\mathcal{T}$. Decidable theories considered are the quantifier-free theory of Presburger arithmetic, congruence closure on ground terms (with or without associative-commutative operators), propositional calculus, and the quantifier-free theory of constructors (mostly, free constructors as in the case of finite lists and finite sequences). A byproduct of the approach is that it can predict the structure of intermediate lemmas needed for automatically deciding this subclass of conjectures. Several examples over lists, numbers and of properties involved in establishing the number-theoretic correctness of arithmetic circuits are given.

⋆ Partially supported by the National Science Foundation Grant nos. CCR-9712396, CCR-9712366, CCR-9996150, and CDA-9503064.

1 Introduction

Inductive reasoning is ubiquitous in verifying properties of computations realized in hardware and software. Automation of inductive reasoning is hampered by the fact that proofs by induction need an appropriate selection of variables for performing induction and a suitable induction scheme, as well as intermediate lemmas. It is well-known that inductive reasoning often needs considerable user guidance, because of which its automation has been a major challenge. A lot of effort has been spent on mechanizing induction in theorem provers (e.g., *Nqthm, ACL2, RRL, INKA, Oyster-Clam*), and induction heuristics in these provers have been successfully used to establish several nontrivial properties. However, the use of induction as a mechanized rule of inference is seriously undermined due to the lack of automation in using this rule. Many reasoning tools including model checkers (in conjunction with decision procedures), invariant generators, deductive synthesis tools preclude induction for lack of automation. This severely limits the reasoning capability of these tools. In many cases inductive properties are established outside these tools manually.

For hardware circuit descriptions, need for inductive reasoning arises when reasoning is attempted about a circuit description parameterized by data width and/or generic components. In protocol verification, induction is often needed when a protocol has to be analyzed for a large set of processors (or network channels). Inductive reasoning in many such cases is not as challenging as in software specifications as well as recursive and loop programs.

This paper is an attempt to address this limitation of these automated tools while preserving their automation, and without having the full generality of a theorem prover. It is shown how decision procedures for simple theories about certain data structures, e.g., numbers, booleans, finite lists, finite sequences, can be enhanced to include induction techniques with the objective that proofs employing such techniques can be done automatically. The result is an extended decision procedure with a built-in induction scheme, implying that an inductive theorem prover can be run in push-button mode as well. We believe the proposed approach can substantially enhance the reasoning power of tools built using decision procedures and model-checkers without losing the advantage of automation.

This cannot be done, however, in general. Conditions are identified on function definitions and conjectures which guarantee such automation. It becomes possible to determine *a priori* whether a given conjecture can be decided automatically, thus predicting the success or failure of using a theorem proving strategy. That is the main contribution of the paper. A byproduct of the proposed approach is that in case of a failure of the theorem proving strategy, it can predict for a subclass of conjectures, the structure of lemmas needed for proof attempts to succeed.

The proposed approach is based on two main ideas: First, terminating recursive definitions of function symbols as rewrite rules oriented using a well-founded ordering, can be used to generate induction schemes providing useful induction hypotheses for proofs by induction; this idea is the basis for the *cover set* method proposed in [16] and implemented in a rewrite-rule based induction theorem prover *Rewrite Rule Laboratory* (*RRL*) [14]. Second, for inductive proofs of conjectures satisfying certain conditions, induction schemes generated from $\mathcal{T}$*-based* recursive function definitions (a concept characterized precisely below), lead to subgoals in $\mathcal{T}$ (after the application of induction hypotheses), where $\mathcal{T}$ is a decidable theory. These conditions are based on the structure of the function definitions and the conjecture.

The concept of a $\mathcal{T}$-*based* function definition is introduced to achieve the above objective. It is shown that conjectures of the form $f(x_1, \cdots, x_m) = r$, where f has a $\mathcal{T}$-based function definition, x_i's are distinct variables, and r is an *interpreted* term in $\mathcal{T}$ (i.e., r includes only function symbols from $\mathcal{T}$), can be decided using the cover set method[3]. The reason for focusing on such simple conjectures is that there is only one induction scheme to be considered by the cover set method. It might be possible to relax this restriction and consider more complicated conjectures insofar as they suggest one induction scheme and the induction hypothesis(es) is applicable to the subgoals after using the function definitions for simplification.

Decidable theories considered are the quantifier-free theory of Presburger arithmetic, congruence closure on ground terms (with or without associative-commutative operators), propositional calculus as well as the quantifier-free theory of constructors (mostly free constructors as in the case of finite lists and finite sequences). For each such theory, decision procedures exist, and *RRL*, for instance, has an implementation of them integrated with rewriting [7, 8].

Below, we review two examples providing an overview of the proposed approach, the subclass of conjectures and definitions which can be considered.

1.1 A Simple Conjecture

Consider the following very simple but illustrative example.

```
(C1): double(m) = m + m,
```

where `double` is recursively defined using the rewrite rules:

```
1. double(0)    --> 0,
2. double(s(x)) --> s(s(double(x))).
```

A proof by induction with `m` as the induction variable and using the standard induction scheme (i.e., Peano's principle of mathematical induction over numbers), leads to one basis goal and one induction step goal. For the basis subgoal, the substitution `m <- 0` gives `double(0) = 0 + 0` which simplifies using the definition of `double` to a valid formula in Presburger arithmetic.

In the step subgoal, the conclusion generated using substitution `m <- s(x)` is `double(s(x)) = s(x) + s(x)`, with the induction hypothesis got by the substitution `m <- x`, being `double(x) = x + x`.

By the second rule in the definition of `double`, the formula simplifies, the induction hypothesis applies, resulting again in a valid formula `s(s(x + x)) = s(x) + s(x)` in Presburger arithmetic. Hence, `(C1)` is valid using Presburger arithmetic and the induction scheme of `double`.

Similarly, a conjecture `double(m) = m` can be decided to be false: the basis case will go through, but the formula resulting from the induction step and the application of the induction hypothesis is not valid.

The main features of the above conjectures and the definition of `double` are:

1. unambiguous induction scheme using which the formula can be decided,

[3] As will be shown later, it is not necessary to require that each argument to f be a distinct variable; instead, non-induction arguments can be interpreted terms that do not include variables in inductive positions of f.

2. induction hypotheses are strong enough to be applicable to induction subgoals, and finally
3. the formulas resulting from subgoals after applying the definition and the induction hypotheses are decidable.

Properties of $\mathcal{T}$-based definitions and simple conjectures ensure the above.

1.2 A Complex Conjecture

For considering complex conjectures including many function symbols with $\mathcal{T}$-based definitions, it becomes necessary to consider the interaction among their definitions based on their nesting order in conjectures. This aspect is captured by the *compatibility* property of function definitions (which is precisely characterized in a later section). The key insight is similar to the one observed of a simple conjecture. Compatible function definitions can be viewed as composing into a single $\mathcal{T}$-based function definition so that a complex conjecture can be viewed as being a simple conjecture in terms of the composed function as illustrated below.

```
(C2): log(exp2(m)) = m.
```

The definitions of the functions `log, exp2` (logarithm and exponentiation to the base 2 respectively) are as follows. Following the mathematical convention, `log` is defined on positive numbers only[4].

```
1. log(s(0))       --> 0,
2. log(x + x)      --> s(log(x)),
3. log(s(x + x))  --> s(log(x)).

4. exp2(0)         --> s(0),
5. exp2(s(x))      --> exp2(x) + exp2(x).
```

Unlike a simple conjecture, the left side of `(C2)` is a nested term. Again, there is only one induction variable `m`, and the induction scheme used for attempting a proof by induction is the principle of mathematical induction for numbers (suggested by the cover set of `exp2`, as explained later in section 2.1).

There is one basis goal and one induction step subgoal. The basis subgoal is `log(exp2(0)) = 0`. The left side rewrites using the definitions of `exp2` and then `log`, resulting in a valid formula in Presburger arithmetic.

In the induction step, the conclusion is `log(exp2(s(x))) = s(x)` with the hypothesis being `log(exp2(x)) = x`. By the definition of `exp2`, `exp2(s(x))` simplifies to `exp2(x) + exp2(x)`. This subgoal will simplify to a formula in Presburger arithmetic if `log(exp2(x) + exp2(x))` rewrites to `s(log(exp2(x)))` either as a part of the definition of `log` or as an intermediate lemma, and then, the induction hypothesis can apply. Such interaction between the definitions of `log` and `exp2` is captured by *compatibility*. Since the definition of `log` includes such a rule, the validity of the induction step case and hence the validity of `(C2)` can be decided.

The validity of a closely related conjecture,

---

[4] This implies that the induction schemes generated using the definition of `log` can be used to decide the validity of conjectures over positive numbers only. For a detailed discussion of the use of cover sets and induction schemes derived from definitions such as `log`, please refer to [9].

```
(C2'): exp2(log(m)) = m,
```

can be similarly decided since `exp2` is compatible with `log`. An induction proof can be attempted using `m` as the induction variable as before. However, `m` can take only positive values since the function `log` is defined only for these. The induction scheme used is different from the principle of mathematical induction. Instead, it is based on the definition of `log`. There is a basis case corresponding to the number `s(0)`, and two step cases corresponding to `m` being a positive even or a positive odd number respectively (this scheme is derived from the cover set of `log` as explained later in section 2.1).

In one of the induction step subgoals, the left side of the conclusion, `exp2(log(s(x + x))) = s(x + x)`, rewrites by the definition of `log` to `exp2(s(log(x)))` which then rewrites to `exp2(log(x)) + exp2(log(x))` to which the hypothesis `exp2(log(x)) = x` applies to produce the inconsistent Presburger arithmetic formula `x + x = s(x + x)`.

As stated above, if `log` and `exp2` are combined to develop the definition of the composed function `log(exp2(x))` from their definitions, then `(C2)` is a simple conjecture about the composed function. Further, the definition of the composed function can be proved to be $\mathcal{T}$-based as well. So the decidability of the conjecture follows.

The notion of compatibility among the definitions of function symbols can be generalized to a *compatible sequence* of function symbols $f_1, \cdots, f_d$ where each f_i is compatible with f_{i+1} at j_i-th argument, $1 \leq i \leq d-1$. A conjecture $l = r$ can then be decided if the sequence of function symbols from the root of l to the innermost function symbol forms a compatible sequence, and r is an interpreted term in $\mathcal{T}$.

The proposed approach is discussed below in the framework of our theorem prover *Rewrite Rule Laboratory* (*RRL*), but the results should apply to other induction provers that rely on decision procedures and support heuristics for selecting induction schemes, e.g., Boyer and Moore's theorem prover *Nqthm*, *ACL2*, and *INKA*. And, the proposed approach can be integrated in tools based on decision procedures and model checking.

The main motivation for this work comes from our work on verifying properties of generic, parameterized arithmetic circuits, including adders, multipliers, dividers and square root [12, 10, 11, 13]. The approach is illustrated on several examples including properties arising in proofs of arithmetic circuits, as well as commonly used properties of numbers and lists involving defined function symbols. A byproduct of this approach is that if a conjecture with the above mentioned restriction cannot be decided, structure of intermediate lemmas needed for deciding it can be predicted. This can aid in automatic lemma speculation.

1.3 Related Work

Boyer and Moore while describing the integration of linear arithmetic into *Nqthm* [3] discussed the importance of reasoning about formulas involving defined function symbols and interpreted terms. Many examples of such conjectures were discussed there. They illustrated how these examples can be done using the interaction of the theorem prover and the decision procedure. In this paper we have focussed on automatically deciding the validity of such conjectures. Most of the examples described there can be automatically decided using the proposed approach.

Fribourg [4] showed that properties of certain recursive predicates over lists expressed as logic programs along with numbers, can be decided. Most of the properties established there can be formulated as equational definitions and decided using the proposed approach. The procedure in [4] used bottom-up evaluation of logic programs which need not terminate if successor operation over numbers is included. The proposed approach does not appear to have this limitation.

Gupta's dissertation [1] was an attempt to integrate (a limited form of) inductive reasoning with a decision procedure for propositional formulas, e.g., ordered BDDs. She showed how properties about a certain subclass of circuits of arbitrary data width can be verified automatically. Properties automatically verified using her approach constitute a very limited subset, however.

2 Cover Set Induction

The cover set method is used to mechanize well-founded induction in *RRL*, and has been used to successfully perform proofs by induction in a variety of nontrivial application domains [12, 10, 11]. For attempting a proof by induction of a conjecture containing a subterm $t = f(x_1, \cdots, x_m)$, where each x_i is a distinct variable, an induction scheme from a complete definition of f given as a set of terminating rewrite rules, is generated as follows. There is one induction subgoal corresponding to each terminating rule in the definition of f. The induction conclusion is generated using the substitution from the left side of a rule, and an induction hypothesis is generated using the substitution from each recursive function call in the right side of the rule. Rules without recursive calls in their right sides lead to subgoals without any induction hypotheses (basis steps).

The recursive definitions of function symbols appearing in a conjecture can thus be used to come up with an induction scheme. Heuristics have been developed and implemented in *RRL*, which in conjunction with failure analysis of induction schemes and backtracking in case of failure, have been found appropriate for prioritizing induction schemes, automatically selecting the "most appropriate" induction scheme (thus selecting induction variables), and generating the proofs of many conjectures.

2.1 Definitions and Notation

Let $T(F, X)$ denote a set of terms where F is a finite set of function symbols and X is a set of variables. A term is either a variable $x \in X$, or a function symbol $f \in F$ followed by a finite sequence of terms, called arguments of f. Let $Vars(t)$ denote the variables appearing in a term t. The *subterms* of a term are the term itself and the subterms of its arguments. A *position* is a finite sequence of positive integers separated by "."'s, which is used to identify a subterm in a term. The subterm of t at the position denoted by the empty sequence ϵ is t itself. If $f(t_1, \cdots, t_m)$ is a subterm at a position p then t_j is the subterm at the position $p.j$. Let $depth(t)$ denote the depth of t; $depth(t)$ is 0 if t is a variable or a constant (denoted by a function symbol with arity 0). $depth(f(t_1, \cdots, t_m)) = maximum(depth(t_i)) + 1$ for $1 \leq i \leq m$.

A term $f(t_1, \cdots, t_m)$ is called *basic* if each t_i is a distinct variable.

A substitution θ is a mapping from a finite set of variables to terms, denoted as $\{x_1 \leftarrow t_1, \cdots, x_m \leftarrow t_m\}$, $m \geq 0$, and $x_i's$ are distinct. θ applied on $s =$

$f(s_1, \cdots, s_m)$ is $f(\theta(s_1), \cdots, \theta(s_m))$. Term s *matches* t under θ if $\theta(s) = t$. Terms s and t *unify* under θ if $\theta(s) = \theta(t)$.

A rewrite rule $s \rightarrow t$ is an ordered pair of terms (s, t) with $Vars(t) \subseteq Vars(s)$. A rule $s \rightarrow t$ is *applicable* to a term u iff for some substitution θ and position p in u, $\theta(s) = u|_p$. The application of the rule *rewrites* u to $u[p \leftarrow \theta(t)]$, the term obtained after replacing the subterm at position p in u by $\theta(t)$. A rewrite system R is a finite set of rewrite rules. R induces a relation among terms denoted $\rightarrow_R$. $s \rightarrow_R t$ denotes rewriting of s to t by a single application of a rule in R. $\rightarrow_R^+$ and $\rightarrow_R^*$ denote the transitive and the reflexive, transitive closure of $\rightarrow_R$.

The set F is partitioned into *defined* and *interpreted* function symbols. An interpreted function symbol comes from a decidable theory $\mathcal{T}$. A defined function symbol is defined by a finite set of terminating rewrite rules, and its definition is assumed to be complete. Term t is *interpreted* if all the function symbols in it are from $\mathcal{T}$. Underlying decidable theories are quantifier-free theories.

A equation $s = t$ is *inductively valid* (valid, henceforth) iff for each variable in it, whenever any ground term of the appropriate type is substituted into $s = t$, the instantiated equation is in the equational theory (modulo the decidable theory $\mathcal{T}$) of the definitions of function symbols in s, t. This is equivalent to $s = t$ holding in the initial model of the equations corresponding to the definitions and the decidable theory $\mathcal{T}$.

Given a complete function definition as a finite set of terminating rewrite rules $\{l_i \rightarrow r_i \mid l_i = f(s_1, \cdots, s_m),\ 1 \leq i \leq k\}$, the main steps of cover set method are

1. *Generating a Cover Set from a Function Definition:* A cover set associated with a function f is a finite set of triples. For a rule $l \rightarrow r$, where $l = f(s_1, \cdots, s_m)$ and $f(t_1^i, \cdots, t_m^i)$ is the i^{th} recursive call to f in the right side r, the corresponding triple is $\langle\langle s_1, \cdots, s_m\rangle, \{\cdots, \langle t_1^i, \cdots, t_m^i\rangle, \cdots\}, \{\}\rangle$[5]. The second component of a triple is the empty set if there is no recursive call to f in r.

   The cover sets of `double`, `exp2` and `log` obtained from their definitions in section 1 are given below.

   ```
   Cover(double):  {<<0>, {}, {}>, <<s(x)>, {<x>}, {}>},
   Cover(exp2):    {<<0>, {}, {}>, <<s(x)>,{<x>}, {}>},
   Cover(log):     {<<s(0)>, {}, {}>,
                    <<x + x>, {<x>}, {}>}, <<s(x + x)>, {<x>}, {}>}.
   ```

2. *Generating Induction Schemes using Cover Sets:* Given a conjecture C, a basic term $t = f(x_1, \cdots, x_m)$ appearing in C can be chosen for generating an induction scheme from the cover set of f. The variables in argument positions in t over which the definition of f recurses are called *induction variables* and the corresponding positions in t are called the *inductive* (or *changeable*) positions; other positions are called the *unchangeable* positions [2].

   An induction scheme is a finite set of induction cases, each of the form $\langle\sigma_c, \{\theta_i\}\rangle$ generated from a cover set triple $\langle\langle s_1, \cdots, s_m\rangle, \{\cdots, \langle t_1^i, \cdots, t_m^i\rangle, \cdots\}, \{\}\rangle$ as fol-

[5] The third component in a triple is a condition under which the conditional rewrite rule is applicable; for simplicity, we are considering only unconditional rewrite rules, so the third component is empty to mean that the rule is applicable whenever its left side matches. The proposed approach extends to conditional rewrite rules as well. See [16, 15, 13].

lows [6]: $\sigma_c = \{x_1 \leftarrow s_1, \cdots, x_m \leftarrow s_m\}$, and , $\theta_i = \{x_1 \leftarrow t_1^i, \cdots, x_m \leftarrow t_m^i\}$.[7]

The induction scheme generated from the cover sets of `double, exp2` is the principle of mathematical induction. The scheme generated from the cover set of `log` is different since the function `log` is defined over positive numbers only. There is one basis step—$\langle\{x \leftarrow s(0)\}, \{\}\rangle$, and $\langle\{x \leftarrow s(0)\}, \{\}\rangle$. There are two induction steps —$\langle\{x \leftarrow m+m\}, \{\{x \leftarrow m\}\}\rangle$, and $\langle\{x \leftarrow s(m+m)\}, x \leftarrow m\rangle$. The variable m is a positive number.

3. *Generating Induction Subgoals using an Induction Scheme:* Each induction case generates an induction subgoal: σ_c is applied to the conjecture to generate the induction conclusion, whereas each substitution θ_i applied to the conjecture generates an induction hypothesis. Basis subgoals come from induction cases whose second component is empty.

   The reader can consult examples `(C1)` and `(C2)` discussed above, and see how induction subgoals are generated using the induction schemes generated from the cover sets of `double` and `exp2`.

3 $\mathcal{T}$-based Definitions and Simple Conjectures

Definition 1. A definition of a function symbol f is $\mathcal{T}$*-based* in a decidable theory $\mathcal{T}$ iff for each rule $f(t_1, \cdots, t_m) \rightarrow r$ in the definition, each t_i, $1 \leq i \leq m$, is an interpreted term in $\mathcal{T}$, any recursive calls to f in r only have interpreted terms as arguments, and the abstraction of r defined as replacing recursive calls to f in r by variables is an interpreted term in $\mathcal{T}$[8].

For examples, the definitions of `double, log` and `exp2` given in Section 1 are $\mathcal{T}$-based over Presburger arithmetic. So is the definition of `*` given using rules,

```
1.  x * 0    --> 0,
2.  x * s(y) --> x + (x * y).
```

We abuse the notation slightly and call the functions themselves as being $\mathcal{T}$-based whenever their definitions are $\mathcal{T}$-based.

In order to use $\mathcal{T}$-based definitions for generating induction schemes, they should be complete as well as terminating over $\mathcal{T}$. For a brief discussion of how to perform such checks, see [9, 6]. It should be easy to see that terms in the cover set generated from a $\mathcal{T}$-based function definition are interpreted in $\mathcal{T}$.

3.1 Simple Conjectures

Definition 2. A term is $\mathcal{T}$*-based* if it contains variables, interpreted function symbols from $\mathcal{T}$ and function symbols with $\mathcal{T}$-based definitions.

[6] The variables in a cover set triple are suitably renamed if necessary.

[7] To generate an induction scheme, it suffices to unify the subterm t in a conjecture C with the left side of each rule in the definition of f as well as with the recursive calls to f in the right side of the rule. This is always possible in case t is a basic term; but it even works if only variables in the induction positions of t are distinct.

[8] If r includes occurrences of `cond`, a special built-in operator in *RRL* for doing simulated conditional rewriting and automatic case analysis, then the first argument to `cond` is assumed to be an interpreted boolean term in $\mathcal{T}$.

Definition 3. A conjecture $f(x_1, \cdots, x_m) = r$, where f has a $\mathcal{T}$-based definition, $x_i's$ are distinct variables and r is interpreted in $\mathcal{T}$, is called *simple.*

Note that both sides of a simple conjecture are $\mathcal{T}$-based.

For example, the conjecture `(C1): double(m) = m + m` about `double` is simple over Presburger arithmetic, whereas the conjecture `(C2): log(exp2(m)) = m` about `log` is not simple over Presburger arithmetic.

For a simple conjecture, the cover set method proposes only one induction scheme, which is generated from the cover set derived from the definition of f.

Theorem 4. *A simple conjecture C over a decidable theory $\mathcal{T}$ can be decided using the cover set method.*

Proof. Given $f(x_1, \cdots, x_m) = r$, where r is interpreted in $\mathcal{T}$, from the cover set associated with the definition of f, an induction scheme can be generated and a proof can be attempted.

Since $\sigma_c(f(x_1, \cdots, x_m)) = l_i$ for some rule $l_i \to r_i$ in the definition of f, the left side of a basis subgoal

$$\sigma_c(f(x_1, \cdots, x_m)) = \sigma_c(r),$$

rewrites using the rule to r_i, an interpreted term in $\mathcal{T}$. The result is a decidable formula in $\mathcal{T}$. This part of the proof exploits the fact that the right side of a simple conjecture, r, is an interpreted term in $\mathcal{T}$.

For each induction step subgoal derived from a rule $l_j \to r_j$ in the definition of f where $r_j = h(\cdots, f(\cdots), \cdots)$, with recursive calls to f, the conclusion is $\sigma_c(f(x_1, \cdots, x_m)) = \sigma_c(r)$; $\sigma_c(f(x_1, \cdots, x_m)) = l_j$ with $\theta_i(f(x_1, \cdots, x_m)) = \theta_i(r)$ being an induction hypothesis corresponding to each recursive call to f in r_j. The left side of the conclusion simplifies by the corresponding rule to r_j which includes an occurrence of $\theta_i(f(x_1, \cdots, x_m))$ as a subterm at a position p_i in r_j. The application of these hypotheses generates the formula $r_j[p_1 \leftarrow \theta_1(r), \cdots, p_k \leftarrow \theta_k(r)] = \sigma_c(r)$ of $\mathcal{T}$, since the abstraction of r_j after recursive calls to f have been replaced by variables, is an interpreted term in $\mathcal{T}$.

Since every basis and induction step subgoal generated by the cover set method can be decided in $\mathcal{T}$, the conjecture C can be decided by the cover set method.□

As the above proof suggests, a slightly more general class of simple conjectures can be decided. Not all the arguments to f need be distinct variables. It suffices if the inductive positions in f are distinct variables, and the other positions are interpreted and do not contain variables appearing in the inductive positions. The above proof would still work.

For example, the following conjecture

```
(C3): append(n, nil) = n,
```

is not simple. The validity of the conjecture `(C3)` can be decided over the theory of free constructors `nil` and `cons` for lists. The definition of `append` is

```
1. append(nil,x)       --> x,
2. append(cons(x,y),z) --> cons(x,append(y,z)).
```

The requirement that unchangeable positions in a conjecture do not refer to the induction variables, seems essential for the above proof to work, as otherwise the application of the induction hypotheses may get blocked.

For example, the cover set method fails to disprove a conjecture such as

```
append(m, m) = m.
```

from the definition of the function `append`. An inductive proof attempt based on the cover set of the function `append` results in an induction step subgoal with the conclusion

```
append(cons(x, y), cons(x, y))  = cons(x, y),
```

and the hypothesis `append(y, y) = y`. The conclusion rewrites to `cons(x, append(y, cons(x, y))) = cons(x, y)` to which the hypothesis cannot be applied. Therefore, the cover set method fails since the induction step subgoal cannot be established.

4 Complex $\mathcal{T}$-based Conjectures

To decide more complex conjectures by inductive methods, the choice of induction schemes have to be limited as well as the interaction among the function definitions have to be analyzed. In [15], such an analysis is undertaken to predict the failure of proof attempts a priori without actually attempting the proof. The notion of **compatibility of function definitions**, an idea illustrated in Section 1, is introduced for characterizing this interaction and for identifying intermediate steps in a proof which get blocked in the absence of additional lemmas.

In this section, we use related concepts to identify conditions under which conjectures such as `(C2)`, more complex than the generalized simple conjectures discussed in section 3, can be decided. We first consider the interaction among two function symbols. This is subsequently extended to consider the interaction among a sequence of function symbols.

Definition 5. A $\mathcal{T}$-based term t is *composed* if

1. t is a basic term $f(x_1, \cdots, x_m)$, where f is $\mathcal{T}$-based and $x_i's$ are distinct variables or
2. (a) $t = f(s_1, \cdots, t', \cdots, s_m)$, where t' is composed and is in an inductive position of a $\mathcal{T}$-based function f, and each s_i is an interpreted term, and
   (b) variables x_i's appearing in the inductive positions of the basic subterm (in the innermost position) of t do not appear elsewhere in t. Other variables in unchangeable positions of the basic subterm can appear elsewhere in t.

For example, the left side of the conjecture `(C2)`, `log(exp2(m))`, is a composed term of depth 2.

The first requirement in the above definition can be relaxed as in the case of simple conjectures. Only the variables in the inductive positions of a basic subterm in t have to be distinct; the terms interpreted in $\mathcal{T}$ can appear in the unchangeable positions of the basic subterm.

Given a conjecture of the form $l = r$,where l is composed and r is interpreted, it is easy to see that there is only one basic subterm in it whose outermost symbol is $\mathcal{T}$-based. The cover set method thus suggests only one induction scheme. We will first consider conjectures such as (C2) in which the left side l is of depth 2; later, conjectures in which l is of higher depth, are considered.

For a conjecture $f(t_1, \cdots, g(x_1, \cdots, x_k), \cdots, t_m) = r$, the interaction between the right sides of rules defining g and the left side of rules defining f must be considered, as seen in the proof of the conjecture (C2). The interaction is formalized below in the property of *compatibility*.

Definition 6. A definition of f is *compatible* with a definition of g in its i-th argument in $\mathcal{T}$ iff for each right side r_g of a rule defining g, the following conditions hold

1. whenever r_g is interpreted, then $f(x_1, \cdots, r_g, \cdots, x_m)$ rewrites to an interpreted term in $\mathcal{T}$, and
2. whenever $r_g = h(s_1, \cdots, g(t_1, \cdots, t_k), \cdots, s_n)$, having a single recursive call to g, the definition of f rewrites $f(x_1, \cdots, h(s_1, \cdots, y, \cdots, s_n), \cdots, x_m)$ to $h'(u_1, \cdots, f(x_1, \cdots, y, \cdots, x_m), \cdots, u_n)$, where x_i's are distinct variables, h, h' are interpreted symbols in $\mathcal{T}$, and s_i, u_j's are interpreted terms of $\mathcal{T}$.[9] In case r_g has many recursive calls to g, say $h(s_1, \cdots, g(t_1, \cdots, t_k), \cdots, g(v_1, \cdots, v_k), \cdots, s_n)$, then the definition of f rewrites $f(x_1, \cdots, h(s_1, \cdots, y, \cdots, z, \cdots, s_n), \cdots, x_m)$ to $h'(u_1, \cdots, f(x_1, \cdots, y \cdots, x_m), \cdots, f(x_1, \cdots, z, \cdots, x_m), \cdots, u_n)$.

The definition of a function f is compatible with a definition of g iff it is compatible with g in every argument position.

As will be shown later, the above requirements on compatibility lead to the function symbol f to be distributed over the interpreted terms to have g as an argument so that the induction hypothesis(es) can be applied.[10]

The above definition is also applicable for capturing the interaction between an interpreted function symbol and a $\mathcal{T}$-based function symbol. For example, the interpreted symbol + in Presburger arithmetic is compatible with * (in both arguments) because of the associativity and commutativity properties of +, which are valid formulas in $\mathcal{T}$.

As stated and illustrated in the introduction, the compatibility property can be viewed as requiring that the composition of f with g has a $\mathcal{T}$-based definition. Space limitations do not allow us to elaborate on this interpretation of compatibility property.

For ensuring the compatibility property, any lemmas already proved about f can be used along with the definition of f. The requirements for showing compatibility can be used to speculate bridge lemmas as well.

The conjecture (C2) is of depth 2. The above insight can be generalized to complex conjectures in which the left side is of arbitrary depth. A conjecture in which a composed term of depth d is equated to an interpreted term, can

[9] The requirement on the definition of f can be relaxed by including bridge lemmas along with the defining rules of f.

[10] In [5], we have given a more abstract treatment of these conditions. The above requirement is one way of ensuring conditions in [5].

be decided if all the function symbols from the root to the position p of the basic subterm in its left side can be pushed in so that the induction hypothesis is applicable. The notion of compatibility of a function definition with another function definition is extended to a *compatible sequence* of definitions of function symbols. In a compatible sequence of function symbols $\langle f_1, \cdots, f_d \rangle$, each f_i is compatible with f_{i+1} at j_i-th argument, $1 \leq i \leq d-1$.

For example, consider the following conjecture

```
(C4): bton(pad0(ntob(m))) = m.
```

Functions `bton` and `ntob` convert binary representations to decimal representations and vice versa, respectively. The function `pad0` adds a leading binary zero to a bit vector. These functions are used to reason about number-theoretic properties of parameterized arithmetic circuits [12, 10]. Padding of output bit vectors of one stage with leading zeros before using them as input to the next stage is common in multiplier circuits realized using a tree of carry-save adders. An important property that is used while establishing the correctness of such circuits is that the padding does not affect the number output by the circuit. The underlying decidable theory is the combination of the quantifier-free theories of bit vectors with free constructors `nil, cons` and `b0, b1`, to stand for binary 0 and 1, and Presburger arithmetic.

In the definitions below, bits increase in significance in the list with the first element of the list being the least significant. Definitions of `bton`, `ntob`, and `pad0` are $\mathcal{T}$-based.

```
 1. bton(nil)           --> 0,
 2. bton(cons(b0, y1)) --> bton(y1) + bton(y1),
 3. bton(cons(b1, y1)) --> s(bton(y1) + bton(y1)),

 4. ntob(0)                --> cons(b0, nil),
 5. ntob(s(0))             --> cons(b1, nil),
 6. ntob(s(s(x2+x2)))      --> cons(b0,ntob(s(x2))),
 7. ntob(s(s(s((x2+x2))))) --> cons(b1,ntob(s(x2))),

 8. pad0(nil)          --> cons(b0, nil),
 9. pad0(cons(b0, y)) --> cons(b0, pad0(y)),
10. pad0(cons(b1, y)) --> cons(b1, pad0(y)).
```

The function `pad0` is compatible with `ntob`; `bton` is compatible with `pad0` as well as `ntob`. However, `ntob` is not compatible with `bton` since `ntob(s(bton(y1) + bton(y1)))` cannot be rewritten using the definition of `ntob`. However, bridge lemmas,

```
ntob(bton(y1) + bton(y1)) = cons(b0, ntob(bton(y1)))
ntob(s(bton(y1) + bton(y1))) = cons(b1, ntob(bton(y1)))
```

can be identified such that along with these lemmas, `ntob` is compatible with `bton`.

A proof attempt of `(C4)` leads to two basis and two step subgoals based on the cover set of `ntob`. The first basis subgoal where `m <- 0`, is

```
bton(pad0(ntob(0))) = 0.
```

The subterm `ntob(0)` rewrites using the definition of `ntob` to `cons(b0, nil)`, then `pad0(cons(b0, nil)))` rewrites to `cons(b0, cons(b0, nil)))`, and finally, `bton(pad0(ntob(0)))` rewrites to `0 + 0 + 0 + 0`, simplifying the above equation to a valid formula in Presburger arithmetic. The second basis subgoal is similar.

Consider the first induction step subgoal. The conclusion is

```
bton(pad0(ntob(s(s(x2 + x2))))) =  s(s(x2 + x2))
```

with the hypothesis being

```
bton(pad0(ntob(s(x2)))) = s(x2).
```

The subterm `ntob(s(s(x2 + x2)))` in the the left side of the conclusion rewrites to `cons(b0, ntob(s(x2)))` by the definition of `ntob`; the subterm `pad0(cons(b0, ntob(s(x2))))` then rewrites to `cons(b0, pad0(ntob(s(x2))))`. Term `bton(pad0(ntob(s(s(x2 + x2)))))` thus rewrites to `bton(pad0(ntob(s(x2)))) + bton(pad0(ntob(s(x2))))`, on which the hypothesis is applicable. The result is a valid formula `s(x2) + s(x2) = s(s(x2 + x2))` in Presburger arithmetic. It can be shown that the second step subgoal also simplifies to a valid formula in Presburger arithmetic.

Every induction subgoal can be decided, and hence `(C4)` can be decided.

The reader would have noticed that the compatibility requirement ensures that all the function symbols are pushed over interpreted symbols for the induction hypothesis to be applicable.

Note: For understanding the proof below, it would be helpful to concurrently consult the proofs of examples `(C2)` in Section 1 as well as of `(C4)` above.

Theorem 7. *The validity of a conjecture $l = r$, where l is a composed term and r is interpreted in $\mathcal{T}$, can be decided by the cover set method if the sequence of function symbols $\langle f_d, f_{d-1}, \cdots, f_2, f_1\rangle$ from the outermost function symbol f_d of l to the basic subterm $f_1(x_1, \cdots, x_m)$ is compatible.*

Proof. By induction on the depth d of l.

Basis case ($d = 2$): Consider a conjecture $l = r$ where $l = f_2(t_1, \cdots, f_1(x_1, \cdots, x_m), \cdots, t_k)$ and $\langle f_2, f_1\rangle$ form a compatible sequence (i.e., f_2 is compatible with f_1 in its argument position), each t_j, $1 \leq j \leq k$, and r are interpreted terms in $\mathcal{T}$. Recall that any induction variable x_i appearing in an inductive position of f_1 does not occur in any t_j.

The cover set method uses the induction scheme generated from the cover set associated with the definition of f_1.

Consider a basis subgoal $\sigma_c(l) = \sigma_c(r)$ generated from a rule $l_1 \rightarrow r_1$ in the definition of f_1, where r_1 does not have any recursive calls to f_1. Since $\sigma_c(f_1(x_1, \cdots, x_m)) = l_1$, $\sigma_c(l)$ rewrites to $f_2(\sigma_c(t_1), \cdots, r_1, \cdots, \sigma_c(t_k))$. Since t_i does not include any induction variable, $\sigma_c(t_i) = t_i$, implying $f_2(\sigma_c(t_1), \cdots, r_1, \cdots, \sigma_c(t_k)) = f_2(t_1, \cdots, r_1, \cdots, t_k)$. Because of compatibility of f_2 with f_1, $f_2(t_1, \cdots, r_1, \cdots, t_k)$ rewrites to an interpreted term. The basis subgoal therefore simplifies to a formula in $\mathcal{T}$.

Consider an induction step subgoal generated from a rule $l_2 \rightarrow r_2$ where $r_2 = h_1(s_1, \cdots, f_1(v_1, \cdots, v_m), \cdots, s_n)$ with a single recursive call to f_1 (for simplicity). Let the conclusion be $\sigma_c(f_2(t_1, \cdots, f_1(x_1, \cdots, x_m) \cdots, t_k)) = \sigma_c(r)$ and the induction hypothesis be $\theta_i(f_2(t_1,\cdots,\ ,\ f_1(x_1,\cdots,x_m),\cdots,t_k)) = \theta_i(r)$, where $\sigma_c(f_1(x_1,\cdots,x_m)) = l_2$ and $\theta_i(f_1(x_1,\cdots,x_m)) = f_1(v_1,\cdots,v_m)$. The left side of the conclusion rewrites to $f_2(t_1,\cdots,r_2,\cdots,t_k)$ (just as in the basis case). As per definition of compatibility of f_2 with f_1, $f_2(y_1,\cdots,h_1(s_1,\cdots,y,\cdots,s_n),\cdots,y_k)$ rewrites to $h_2(s'_1,\cdots,f_2(y_1,\cdots,y,\cdots,y_k),\cdots,s'_n)$ where y_i and y are distinct variables, and h_2, s_j's and s'_j's are in $\mathcal{T}$. This means that the left side of the simplified conclusion $f_2(t_1,\cdots,r_2,\cdots,t_k)$ rewrites by the same sequence of rules to $h_2(\delta(s'_1),\cdots,\delta(f_2(y_1,\cdots,y,\cdots,y_k)),\cdots,\delta(s'_n))$, where $\delta(y_i) = t_i, 1 \leq i \leq k$, and $\delta(y) = f_1(v_1,\cdots,v_m)\}$. The hypothesis applies since $\theta_i(t_j) = t_j$ for all t_j (recall that there are no x_i's in t_j's), and $\theta_i(f_1(x_1,\cdots,x_m)) = f_1(v_1,\cdots,v_m)$, which simplifies the conclusion to $h_2(\delta(s'_1),\cdots,\theta_i(r),\cdots,\delta(s'_n)) = \sigma_c(r)$, a formula in $\mathcal{T}$.

The above proof step assumed a single recursive call in r_2 and the application of a single induction hypothesis. The proof generalizes when there are multiple recursive calls in r_2 and many possibly different hypotheses have to be applied.

Induction Step case: Assume that the statement of the theorem for all conjectures $l' = r'$, where l' is a composed term of depth $d' < d$, and r' is interpreted.

The main idea in this proof is to use the fact that a conjecture $l = r$ in which the composed term $l = f_d(t_1,\cdots,l_{d-1},\cdots,t_k)$ is of depth d, uses the same induction scheme as a related conjecture $l_{d-1} = c$, where l_{d-1} is a composed term of depth $d-1$, and c is an interpreted term. By the induction hypothesis, $l_{d-1} = c$ can be decided since all subgoals, including basis and induction steps, can be decided. Because of the compatibility of f_d with f_{d-1}, the outermost symbol of l_{d-1}, it can be shown that each subgoal of $l = r$ using the same induction scheme can also be decided. In the basis step, the instantiated conjecture rewrites to a formula in $\mathcal{T}$, and in the induction step, f_d can be pushed over the interpreted symbols to surround f_{d-1} so that the hypothesis is again applicable, resulting in a formula in $\mathcal{T}$. More details follow.

The same basic term $f_1(x_1,\cdots,x_k)$ in l_{d-1} used for generating an induction scheme for $l_{d-1} = c$ is also used for generating an induction scheme for $l = r$. By the induction hypothesis, each of the subgoals generated from $l_{d-1} = c$ using this induction scheme can be decided in $\mathcal{T}$. For $l = r$ as well, a subgoal $\sigma_c(f_d(t_1,\cdots,l_{d-1},\cdots,t_k)) = \sigma_c(r)$ using the same substitution can be decided.

Consider a basis subgoal $\sigma_c(l_{d-1}) = \sigma_c(c)$ where $\sigma_c(l_{d-1})$ simplifies to the interpreted term u through a sequence of rewrite steps using the definitions of the $\mathcal{T}$-based function symbols in l_{d-1}. (The $\mathcal{T}$-based function symbols in l_{d-1} are successively eliminated in a bottom up fashion starting with f_1 until finally f_{d-1} rewrites to u by a rule of the form $f_{d-1}(\cdots,r_g,\cdots) \to u$ in the definition of f_{d-1}.) By the compatibility of f_d with f_{d-1}, $f_d(t_1,\cdots,u,\cdots,t_k)$ rewrites to an interpreted term, say u', implying that the basis subgoal $\sigma_c(l) = \sigma_c(r)$ simplifies to $u' = \sigma_c(r)$, a formula in $\mathcal{T}$.

Consider an induction step subgoal with the conclusion $\sigma_c(l_{d-1}) = \sigma_c(c)$ and the hypothesis $\theta_i(l_{d-1}) = \theta_i(c)$, generated from a rule in the definition of f_1 whose right side has a single recursive call to f_1 (for simplicity). The left side of the conclusion simplifies through a sequence of rewrite steps using the definitions of the $\mathcal{T}$-based function symbols in l_{d-1} to a term of the form $h_{d-1}(s'_1,\cdots,\theta_i(l_{d-1}),\cdots s'_n)$ where $s'_j s$ and h_{d-1} are interpreted. The $\mathcal{T}$-based functions in l_{d-1} are successively pushed over interpreted function symbols in a bottom up fashion until finally f_{d-1} is pushed using a rule of the form $f_{d-1}(y_1,h_{d-2}(s_1,\cdots,y,\cdots,s_n),\cdots,y_k) \to h_{d-1}(s'_1,\cdots,f_{d-1}(y_1,\cdots,x,\cdots,y_k),\cdots,s'_n)$, to get the left side of the hypothesis. By compatibility of f_d with f_{d-1}, $f_d(z_1,\cdots,h_{d-1}(s'_1,\cdots,z,\cdots,s'_n)\cdots,z_k)$ rewrites to $h_d(s''_1,\cdots,f_d(z_1,\cdots,z,\cdots,z_k),\cdots,s''_n)$. where h_d and $s''_j{}'s$

are interpreted. This implies that in the corresponding induction step subgoal, the left side of the conclusion $\sigma_c(f_d(t_1, \cdots, l_{d-1}, \cdots, t_k))$ will simplify to $h_d(s_1'' \cdots, \theta_i(f_d(t_1, \cdots l_{d-1}, \cdots, t_k)), \cdots, s_n'')$, a term containing the left side of the hypothesis. The application of the hypothesis simplifies the conclusion to $h_d(s_1'' \cdots, \theta_i(r), \cdots, s_n'') = \sigma_c(r)$, a formula in $\mathcal{T}$. □

5 Relaxing Linearity Requirement: Nonlinear Conjectures

To cover a larger class of formulas, we discuss conditions for deciding a conjecture with multiple occurrences of induction variables in its left side.

Definition 8. A conjecture $f(s_1, \cdots, s_m) = r$, where $f(s_1, \cdots, s_m)$ is a $\mathcal{T}$-based term, r is interpreted in $\mathcal{T}$, and for $1 \leq i \leq m$, either s_i is interpreted in $\mathcal{T}$, or $s_i = g_i(x_1, \cdots, x_n)$ is a basic term, is called *basic nonlinear* if some variable has multiple occurrences in l.

In a basic nonlinear conjecture, induction variables (as well as noninduction variables) appearing as arguments in basic terms can be shared. For example, the conjecture below is basic nonlinear,

```
(C5): append(blast(m), last(m)) = m,
```

where `last` returns the singleton list containing the last element of a list, and `blast` returns the input list without the last element.

```
1. last(cons(x, nil))          --> cons(x, nil).
2. last(cons(x, cons(y, z))) --> last(cons(y, z)),
3. blast(cons(x,nil))          --> nil,
4. blast(cons(x,cons(y,z))) --> cons(x,blast(cons(y,z))).
```

To decide such a conjecture, additional conditions become necessary. First, since there can be many induction schemes possible, one each generated from the cover set of a basic term, it is required that they can be merged into a single induction scheme [2, 9] (the case when each cover set generates the same induction scheme trivially satisfies this requirement). The second requirement is similar to that of compatibility: f above must be *simultaneously compatible* with each g_i. In the definition below, we assume, for simplicity, that there is at most one recursive call in the function definitions.

Definition 9. The definition of f is *simultaneously compatible* in $\mathcal{T}$ with the definitions of g and h in its i^{th} and j^{th} arguments, where $i \neq j$ if for each right side r_g and r_h of the rules in the definitions of g and h, respectively:

1. whenever r_g and r_h are interpreted in $\mathcal{T}$, $f(x_1, \cdots, r_g, \cdots, r_h, \cdots, x_m)$ rewrites to an interpreted term in $\mathcal{T}$, and
2. whenever $r_g = h_1(\cdots, g(\cdots), \cdots)$ and $r_h = h_2(\cdots, h(\cdots), \cdots)$, the definition of f rewrites $f(x_1, \cdots, h_1(\cdots, x, \cdots), \cdots, h_2(\cdots, y, \cdots), \cdots, x_m)$ to $h_3(\cdots, f(x_1, \cdots, x, \cdots, y, \cdots, x_m), \cdots)$.

For example, `append` is simultaneously compatible with `blast` in its first argument and `last` in its second argument.

Theorem 10. *A basic nonlinear conjecture* $f(\cdots, g(x_1, \cdots, x_n), \cdots, h(x_1, \cdots, x_n), \cdots) = r$, *such that* $x_1, \cdots, x_n$ *do not appear elsewhere in the left side of the conjecture and the remaining arguments of* f *are interpreted terms, can be decided by the cover set method if* f *is simultaneously compatible with* g *and* h *at* i^{th} *and* j^{th} *arguments, respectively, and the induction schemes suggested by* $g(x_1, \cdots, x_n)$ *and* $h(x_1, \cdots, x_n)$ *can be merged.*

The proof is omitted due to lack of space; it is similar to the proof of the basis case of Theorem 7. The main steps are illustrated using a proof of `(C5)`.

The induction schemes suggested by the basic terms `blast(m)`, `last(m)` in `(C5)` are identical. There is one basis subgoal and one induction step subgoal. The basis subgoal obtained by `m <- cons(x, nil)`,

```
append(blast(cons(x, nil)), last(cons(x, nil))) =  cons(x, nil),
```

simplifies to a valid formula by the definitions of `blast` and `last`, and then by the definition of `append`.

In the step subgoal, the induction conclusion is

```
append(blast(cons(x,cons(y,z))),last(cons(x,cons(y,z)))) = cons(x,cons(y,z)),
```

with the hypothesis being,

```
   append(blast(cons(y, z)), last(cons(y, z))) = cons(y, z).
```

The left side of the conclusion rewrites by the definitions of `last`, `blast` to `append(cons(x, blast(cons(y, z))), last(cons(y, z))))` which rewrites using the definition of `append` to `cons(x, append(blast(cons(y, z)), last(cons(y, z))))` to which the hypothesis applies, leading to the valid formula `cons(x, cons(y, z)) = cons(x, cons(y, z))`.

The notion of simultaneous compatibility and the above theorem generalize to complex nonlinear conjectures, similar to the complex conjecture `(C4)` discussed in Section 4, in which a conjecture includes a sequence of simultaneously compatible function symbols. Because of space limitations, we cannot discuss this in detail here. The example below illustrates the idea to some extent. The underlying theory is that of free constructors with `0, s`. The function symbol `+` is assumed to have the usual recursive definition: `0 + y --> y, s(x) + y --> s(x + y)`. The equation is:

```
(C6):  mod2(x) + (half(x) + half(x)) = x,
```

is a complex nonlinear conjecture with the following definitions of `half` and `mod2`.

```
1. half(0)        --> 0,
2. half(s(0))     --> 0,
3. half(s(s(x))) --> s(half(x)).

4. mod2(0)        --> 0,
5. mod2(s(0))     --> s(0),
6. mod2(s(s(x))) --> mod2(x).
```

For `+` to be compatible with `half` in both its arguments, an intermediate lemma (either the commutativity of `+` or `x + s(y) = s(x + y)`) is needed as well.[11]

It can be a priori determined that `(C6)` can be decided by the cover set method since the basic terms `half(x), mod2(x)` suggest the same induction scheme, and the function symbol `+` is simultaneously compatible with `mod2, +` as well as `half)` in the presence of the above lemma about `+`.

6 Bootstrapping

As discussed above, simple and complex conjectures with $\mathcal{T}$-based function symbols can be decided using the cover set method, giving an extended decision procedure and an extended decidable theory. In this section, we outline preliminary ideas for bootstrapping this extended decidable theory with the definitions of $\mathcal{T}$-based function symbols and the associated induction schemes, to define and decide a larger class of conjectures.

Definition 11. A definition of a function symbol f is *extended $\mathcal{T}$-based* for a decidable theory $\mathcal{T}$ if for each rule, $f(t_1, \cdots t_m) \to r$ in the definition, where $t_i's$ are interpreted over $\mathcal{T}$, the only recursive call to f in r, if any, has only $\mathcal{T}$-based terms as arguments, and the abstraction of r after replacing the recursive call to f by a variable, is either an interpreted term over $\mathcal{T}$, or a basic term $g(\cdots)$ where g has an (extended) $\mathcal{T}$-based definition.

For example, `exp` denoting exponentiation, defined below, is extended $\mathcal{T}$-based over Presburger arithmetic. For rules defining `*`, please refer to the beginning of Section 3.

```
1. exp(x, 0)    --> s(0),
2. exp(x, s(y)) --> x * exp(x, y).
```

Unlike simple conjectures, an inductive proof attempt of an extended $\mathcal{T}$-based conjecture may involve multiple applications of the cover set method. Induction may be required to decide the validity of the induction subgoals. In order to determine a priori this, the number of recursive calls in any rule in an extended $\mathcal{T}$-based definition, is restricted to be at most one. The abstracted right side r could be an interpreted term in $\mathcal{T}$, or a basic term with an extended $\mathcal{T}$-based function.

Theorem 12. *A simple extended $\mathcal{T}$-based conjecture $f(x_1, \cdots, x_m) = r$, where f is an extended $\mathcal{T}$-based function, and r is interpreted over $\mathcal{T}$, can be decided by the cover set method.*

The key ideas are suggested in the disproof of an illustrative conjecture about `exp`:

```
(C7): exp(s(0), m)   = s(m).
```

In the proof attempt of `(C7)`, with induction variable `m`, there is one basis and one step subgoal. The basis subgoal,

[11] If `+` is defined by recursing on the second argument, even then commutativity of `+` or `s(x) + y = s(x + y)` is needed.

```
exp(s(0), 0) = s(0)
```

rewrites by definition of `exp` to the valid formula `s(0) = s(0)`. In the step subgoal, the conclusion

```
exp(s(0), s(y))  =  s(s(y)),
```

rewrites by definition of `exp` to `s(0) * exp(s(0), y) =  s(s(y))`, to which the hypothesis, `exp(s(0), y) = s(y)`, applies to give `s(0) * s(y) =  s(s(y))`, which then rewrites by definition of `*` to `s(0) * y =  s(y)`, a simple conjecture which can be decided to be false by the cover set method.

Complex extended $\mathcal{T}$-based conjectures can be similarly defined, and conditions for deciding their validity can be developed. This is currently being explored.

7 Conclusion

This paper describes how inductive proof techniques implemented in existing theorem provers, such as *RRL*, can be used to decide a subclass of equational conjectures. Sufficient conditions for such automation are identified based on the structure of the conjectures and the definitions of the function symbols appearing in the conjectures as well as interaction among the function definitions.

The basic idea is that if the conditions are met, the induction subgoals automatically generated a conjecture by the cover set method simplify to formulas in a decidable theory. This is first shown for simple conjectures with a single function symbol recursively defined using interpreted terms in a decidable theory. Subsequently, this is extended to complex conjectures with nested function symbols by defining the notion of compatibility among their definitions. The compatibility property ensures that in induction subgoals, function symbols can be pushed inside the instantiated conjectures using definitions and bridge lemmas, so as to enable the application of the induction hypotheses, leading to decidable subgoals.

It is shown that certain nonlinear conjectures with multiple occurrences of induction variables can also be decided by extending the notion of compatibility to that of simultaneous compatibility of a function symbol to many function symbols. Some preliminary ideas on bootstrapping the proposed approach are discussed by considering conjectures with function symbols that are defined in terms of other recursively defined function symbols.

Our preliminary experience regarding the effectiveness of the proposed conditions is encouraging. Several examples about properties of lists and numbers as well as properties used to establish the number-theoretic correctness of arithmetic circuits have been successfully tried.

Some representative conjectures, both valid and nonvalid formulas, decided by the proposed approach are given below. With each conjecture, the annotations indicate whether it is simple or complex, as discussed above, its validity and the underlying decidable subtheories. Conjectures are annotated as being nonlinear if they contain multiple basic terms with the same induction variables. For example, the conjectures 12-16 below are nonlinear since they have multiple basic terms with the induction variable `x`. However, conjectures are 7-9 are not nonlinear since they do not contain multiple basic terms even though they contain multiple occurrences of the variable `x`. In conjectures 18-20, the underlying theory is Presburger arithmetic extended with the function symbol `*`.

Conjectures 16 and 17 establish the correctness of a restricted form of ripple-carry and carry-save adders respectively. The arguments to the two adders are restricted to be the same in these conjectures. This restriction can be relaxed, and the number-theoretic correctness of parameterized ripple-carry and carry-save adders[12, 10] can be done using the proposed approach. In addition, several intermediate lemmas involved in the proof of multiplier circuits and the SRT divider circuit [10, 11] can be handled.

```
 1. half(double(x))       = x,         [Complex, valid, Presburger]
 2. mod2(double(x))       = 0,         [Complex, valid, Presburger]
 3. half(mod2(x))         = 0,         [Complex, valid, Presburger]
 4. log(mod2(x))          = 0,         [Complex, valid, Presburger]
 5. exp2(log(x))          = x,         [Complex, inval, Presburger]
 6. log(exp2(x))          = x,         [Complex, valid, Presburger]
 7. x * log(mod2(x)))     = 0,         [Complex, valid, Presburger]
 8. x * mod2(double(x))   = 0,         [Complex, valid, Presburger]
 9. memb(x, delete(x, y)) = false,     [Complex, valid, lists]
10. bton(pad0(ntob(x)))   = x,         [Complex, valid, lists]
11. last(ntob(double(x))) = 0,         [Complex, valid, lists]
12. length(append(x, y)) - (length(x) + length(y)) = 0,
                              [Complex, valid, nonlinear, Presburger, Lists]
13. rotate(length(x), x)  = x,         [Complex,valid,nonlinear,Presburger,Lists]
14. length(nth(x, y))    <= length(x), [Complex,valid,nonlinear,Presburger,Lists]
15. length(delete(x, y)) <= length(y), [Complex,valid,nonlinear,Presburger,Lists]
16. bton(carry-saveadder(ntob(x), ntob(x), ntob(x)))   = x + x + x,
                              [Complex, valid, nonlinear, Presburger, Bitvectors]
17. bton(ripple-carryadder(ntob(x), ntob(x), ntob(x))  = x + x + x.
                              [Complex, valid, nonlinear, Presburger, Bitvectors]
18. exp(1, x)            = x,          [Simple, valid, Presburger extend by *]
19. exp(1, x)            = s(x),       [Simple, inval, Presburger extend by *]
20. exp(x, mod2(double(y))) = s(0)     [Complex,valid, Presburger extend by *]
```

Inductive reasoning plays a central role in several nontrivial applications, but induction techniques are hardly supported in many reasoning tools, primarily due to the intense manual intervention required to perform inductive proofs in general. The proposed approach can be used to integrate induction proof methods in other reasoning tools and selectively invoke these methods to significantly enhance the reasoning capabilities of these tools without compromising automation. For instance, procedures implementing the cover set induction method can be integrated as a component decision procedure in a cooperating decision procedures framework; it can be invoked to check the validity of inductive subgoals.

To make the proposed approach more effective, it should be generalized to decide more general quantifier-free formulas as well as mechanically generate subsidiary conditions under which a given quantifier-free formula is valid. Such an investigation has been initiated and preliminary results are discussed in [5]. It is also necessary to consider decidability of formulas that require nested induction. Another promising direction for extending this work is to use the proposed approach to guide generation of intermediate lemmas.

Acknowledgements: Thanks to Juergen Giesl and the referees for useful comments on an earlier draft of the paper.

References

1. A. Gupta, *Inductive Boolean Function Manipulation: A Hardware Verification Methodology for Automatic Induction.* Ph.D. Thesis, Department of Computer Science, Carnegie Mellon University, Pittsburgh, 1994.
2. R.S. Boyer and J S. Moore, *A Computational Logic.* ACM Monographs in Computer Science, 1979.
3. R.S. Boyer and J S. Moore, "Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic", *Machine Intelligence* 11, J.E. Hayes, D. Mitchie and J. Richards (eds), 1988.
4. L. Fribourg, "Mixing list recursion and arithmetic", Proc. *Seventh Symp. on Logic in Computer Science*, 1992.
5. J. Giesl and D. Kapur, *Decidable Classes of Inductive Theorems.* Technical Report, Department of Computer Science, University of New Mexico, Feb. 2000.
6. D. Kapur, "Automated tools for analyzing completeness of specifications," Proc. *1994 Intl. Symp. on Software Testing and Analysis (ISSTA)*, Seattle, WA, August 1994, 28-43.
7. D. Kapur, "Rewriting, decision procedures and lemma speculation for automated hardware verification," Proc. *10th Intl. Conf. Theorem Proving in Higher Order Logics,* LNCS 1275, 1997.
8. D. Kapur and X. Nie, "Reasoning about numbers in Tecton," Proc. *8th Intl. Symp. Methodologies for Intelligent Systems, (ISMIS'94)*, North Carolina, October 1994.
9. D. Kapur and M. Subramaniam, "New uses of linear arithmetic in inductive theorem proving," *J. Automated Reasoning,* 16 (1-2), 1996, 39-78.
10. D. Kapur and M. Subramaniam, "Mechanically verifying a family of multiplier circuits," Proc. *Computer Aided Verification (CAV'96)*, New Jersey, Springer LNCS 1102 (eds. Alur & Henzinger), 1996, 135-146.
11. D. Kapur and M. Subramaniam "Mechanizing reasoning about arithmetic circuits: SRT division," Proc. *17th FSTTCS,* LNCS (eds. Sivakumar & Ramesh), 1997.
12. D. Kapur and M. Subramaniam, "Mechanical verification of adder circuits using powerlists," *J. of Formal Methods in System Design,* Nov. 1998.
13. D. Kapur and M. Subramaniam, "Using an induction prover for verifying arithmetic circuits," to appear in *J. of Software Tools for Technology Transfer,* Springer Verlag, March 1999.
14. D. Kapur, and H. Zhang, "An overview of Rewrite Rule Laboratory (RRL)," *J. of Computer and Mathematics with Applications,* 29, 2, 1995, 91-114.
15. M. Subramaniam, *Failure Analyses in Inductive Theorem Provers.* Ph.D. Thesis, Department of Computer Science, University at Albany, State University of New York, 1997.
16. H. Zhang, D. Kapur, and M.S. Krishnamoorthy, "A mechanizable induction principle for equational specifications," Proc. *9th Intl. Conf. Automated Deduction (CADE),* Springer LNCS 310, (eds. Lusk and Overbeek), Chicago, 1988, 250-265.

This article was processed using the LaTeX macro package with LLNCS style