Automatic Generation of Simple Lemmas from Recursive Definitions using Decision Procedures[⋆] — Preliminary Report —

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

[1] Department of Computer Science, University of New Mexico, Albuquerque, NM
kapur@cs.unm.edu
[2] Department of Computer Science, University of Nebraska, Omaha, NE
msubramaniam@mail.uomaha.edu

Abstract. Using recent results on integrating induction schemes into decidable theories, a method for generating lemmas useful for reasoning about $\mathcal{T}$-based function definitions is proposed. The method relies on terms in a decidable theory admitting a (finite set of) canonical form scheme(s) and ability to solve parametric equations relating two canonical form schemes with parameters. Using nontrivial examples, it is shown how the method can be used to automatically generate many simple lemmas; these lemmas are likely to be found useful in automatically proving other nontrivial properties of $\mathcal{T}$-based functions, thus unburdening the user of having to provide many simple intermediate lemmas. During the formalization of a problem, after a user inputs $\mathcal{T}$-based definitions, the method can be employed in the background to explore a search space of possible conjectures which can be attempted, thus building a library of lemmas as well as false conjectures. This investigation was motivated by our attempts to automatically generate lemmas arising in proofs of generic, arbitrary data-width parameterized arithmetic circuits. The scope of applicability of the proposed method is broader, however, including generating proofs for proof-carrying codes, certification of proof-carrying code as well as in reasoning about distributed computation algorithms.

1 Introduction

A major challenge in automating proofs of inductive properties is the need to generate intermediate lemmas typically needed in such proofs [2, 3, 15]. Often, many lemmas needed in such proofs are simple properties of functions appearing in an original conjecture and they can be easily proved once they can be properly formulated. Inability to speculate intermediate lemmas is one of the reasons, we suspect, for induction theorem provers not being used by domain experts in their applications including verification of hardware circuits and distributed algorithms, certification of proof-carrying codes, analysis of design specifications, etc. For a novice user of an induction theorem prover, such intermediate lemmas are hard to formulate, since failed proof attempts of the original application specific conjecture have to be analyzed manually. In order to do this successfully, the application expert has to become an expert in understanding the internal representation of proofs, how proofs are generated by theorem provers, and various heuristics employed to generate such proofs. And, this can be tedious and time consuming. This demand can become an added burden on the application user since most such properties are a result of a particular formalization being attempted, and may have little to do directly with the application.

[⋆] This research was partially supported by an NSF ITR award CCR-0113611.

The use of theorem provers can be made more effective if simple and seemingly obvious properties of functions employed in the formalization of application specific properties can be automatically attempted. This can provide immediate feedback to the user about the functions used in the formalization thus either leading to fixes in the formalization or enhancing confidence in the formalization.

In this paper, a method is proposed using decision procedures for speculating and proving "simple" properties of functions likely to be found useful soon after the function definitions are introduced. The proposed approach relies on the structure of $\mathcal{T}$-based function definitions introduced in [11] and properties of decision procedures for decidable theories $\mathcal{T}$. Below, we briefly discuss the main ideas underlying the proposed approach.

This line of research was motivated by our attempts to automatically generate lemmas arising in proofs of generic, arbitrary parameterized arithmetic circuits. The scope of applicability of the proposed method is broader, however, including generating proofs for proof-carrying codes, certification of proof-carrying code as well as in reasoning about distributed computation algorithms.

1.1 Overview

Given a definition of f, the main idea is to hypothesize a conjecture of the form $f(x_1, \cdots, x_k) = r$, where r is a unknown (parameterized) term in a decidable theory $\mathcal{T}$ possibly involving $x_1, \cdots, x_k$ as well as parameters p_i's whose values need to be determined for all possible values of x_j's. In fact, r is one of the many possible canonical forms of a term in $\mathcal{T}$. As shown in [11], such a conjecture can be decided if f is $\mathcal{T}$-based. A proof is attempted of this conjecture, possibly leading to instantiation of various parameters. If all the parameters can be consistently instantiated, then the right hand side r of the conjecture can be determined, thus implying that f is expressible in $\mathcal{T}$.

For example, consider the function *cost* denoting a recurrence relation for a divide and conquer algorithm.

1. $cost(0) \rightarrow 0$, 2. $cost(1) \rightarrow 2$,
3. $cost(s(x) + s(x)) \rightarrow cost(x) + cost(x) + 4$. 4. $cost(s(s(x) + s(x))) \rightarrow cost(x) + cost(x) + 6$.

As discussed later (also see [11]) in section 2, the above definition can be shown to be based in the quantifier-free theory of Presburger arithmetic, denoted by PA[1]. It may be speculated that $cost(m)$ is expressible in PA and is thus equivalent to a term in PA, say $k_1\ m + k_2$ with k_1, k_2 as its parameters (without any loss of generality); $k_1\ m$ stands for m repeated k_1 times. So we generate a parametric conjecture $P1 : cost(m) = k_1\ m + k_2$ with the goal of finding k_1, k_2.

An induction proof using the cover set method [17] using the definition of *cost* is attempted. The first subgoal generated by unifying the left side of rule 1 with the left side of *P1* with substitution $\{m \rightarrow 0\}$ is $cost(0) = k_1\ 0 + k_2$, which simplifies using rule 1 to the constraint $s1 : k_2 = 0$. The second subgoal generated by unifying the left side of rule 2 with the left side of *P1* with substitution $\{m \rightarrow 1\}$ is $cost(1) = k_1 + k_2$, which simplifies using rule 2 to the constraint $s2 : k_1 + k_2 = 2$. In the third subgoal, the conclusion generated by unifying the left side of rule 3 with the left side of *P1* with substitution $\{m \rightarrow s(x) + s(x)\}$ is $cost(s(x) + s(x)) = k_1\ x + k_1 + k_1\ x + k_1 + k_2$, which simplifies using rule 3 to $cost(x) + cost(x) + 4 = k_1\ x + k_1 + k_1\ x + k_1 + k_2$, Replacing $cost(x)$ by $k_1\ x + k_2$, one gets $k_1\ x + k_2 + k_1\ x + k_2 + 4 = k_1\ x + k_1 + k_1\ x + k_1 + k_2$ which further simplifies

[1] The theory PA includes $0, s, +, =$; the symbols $1, 2, 3$, etc., stand for $s(0)$, $s(s(0))$, $s(s(s(0)))$, respectively.

to $s3 : k_2 + 4 = k_1 + k_1$. Similarly, from the fourth subgoal generated using rule 4, we get $s4 : k_2 + 6 = k_1 + k_1 + k_1$.

Solving for the constraints $s1$, $s2$, $s3$, $s4$ (using a decision procedure for PA) gives $k_1 = 2, k_2 = 0$; thus, $cost(m) = 2\ m$, implying that $cost$ is expressible in PA [2].

Typically, it will be not be possible to express a recursively defined function in $\mathcal{T}$ since new recursive definitions are often introduced because the functions being defined cannot be expressed in $\mathcal{T}$. (If a defined function can be expressed in $\mathcal{T}$, that is more likely to be an error in the definition than the user's improper grasp of the expressive power of the theory.) What is more likely is that in general, parameters cannot be consistently instantiated; instead, for some specific values of x_i's, parameter values can be obtained, thus implying that for those values of x_i's, the instantiated conjecture equals some term in $\mathcal{T}$.

For example, consider the following definition of $e2plus$:

1. $e2plus(0, 0) \rightarrow 1$,
2. $e2plus(0, s(y)) \rightarrow e2plus(0, y) + 1$,
3. $e2plus(s(x), 0) \rightarrow e2plus(x, 0) + e2plus(x, 0)$,
4. $e2plus(s(x), s(y)) \rightarrow s(e2plus(s(x), y))$.

Similar to the previous example, it may be speculated that $e2plus$ can be expressed in PA and a conjecture $P2 : e2plus(x, y) = k_1\ x + k_2\ y + k_3$ may be hypothesized with the goal of finding k_1, k_2, k_3.

To solve for the parameters, constraints are generated from the rules 1-4. The first constraint $s1 : k_3 = 1$ comes from rule 1. The second constraint is generated from rule 2 by replacing the left side and the recursive call with $k_1\ 0 + k_2\ s(y) + k_3$ 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, primarily because of $s3$ which restricts $k_1 = 0$ and hence $k_3 = 0$, whereas the constraint $s1$ gives $k_3 = 1$. 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, and (ii) constraints involving both parameters and variables (e.g. $s3$).

Now conjectures with proper instances of $e2plus(x, y)$ as their left 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 (see subsection 3.2 for details). One such maximal consistent subset 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 $\{s1, s2, s4\}$[3]. The rules associated with the instances are computing by splitting the rule using the variable substitution (see subsection 2.3 for details), e.g:, rule 3 is split by $\{x \rightarrow 0\}$ into two rules,

3.1 $e2plus(s(0), 0) \rightarrow 2$,
3.2 $e2plus(s(s(x)), 0) \rightarrow e2plus(x, 0)\ +\ e2plus(x, 0)\ +\ e2plus(x, 0) + e2plus(x, 0)$.

The constraints for rule 3.1 is $s3.1' : k_1 + k_3 = 2$ and rule 3.2 is $s3.2 : 2k_1 = 3k_1\ x + 3k_3$.

The rules to be considered for speculating a conjecture are $\{1, 2, 3.1, 4\}$. From the left side of these rules, a possible conjecture with $e2plus(0, y)$ as the left side is generated (since the left sides of rules 1 and 2 are $e2plus(0, 0), e2plus(0, s(y))$) (see subsection 2.4 for details).

[2] In almost all cases, it is possible to use the right side of the conjecture in the parametric form to substitute in each of the rules in the definitions and directly get a constraint. The cover set induction method is being used just as a general purpose mechanism to do the same.

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

The rules $\{1, 2, 3.1, 3.2, 4\}$ are analyzed to pick their instances which are likely to be used to compute $e2plus(0, x)$. Rules 1 and 2 constitute such a complete set since any ground instance of $e2plus(0, x)$ can be normalized to a PA-term using them (see subsection 3.3 for details). The constraints generated from rules 1 and 2 have the solution $\{k_2 = 1, k_3 = 1\}$, which generates the lemma $e2plus(0, x) = s(x)$. We make it rule 5 and add it to the definition of $e2plus$.

$$5. \quad e2plus(0, y) \rightarrow s(y).$$

This lemma can replace rules 1 and 2, and can be used to speculate additional lemmas from rules 3, 4 and 5 which completely define $e2plus$.

Rules associated with the maximal set $\{1, 2, 3.1, 4\}$ may also be expanded to generate additional rules for generating left sides for conjectures (see subsection 3.3 for details). The expansion is done by choosing a rule whose right side is a PA-term, e.g: rule 3.1, and splitting by unifying the recursive calls in other rules with the left side of rule 3.1. The unification of the recursive call $e2plus(s(x), y)$ in rule 4 with $e2plus(s(0), 0)$ of rule 3.1 produces four equivalent rules two of which are:

$$4.1\ e2plus(s(0), s(0)) \rightarrow 3, \qquad 4.2\ e2plus(s(0), s(s(y))) \rightarrow s(s(e2plus(s(0), y))).$$

The conjecture with the left side $e2plus(s(0), y)$ can now be hypothesized by abstracting from the left sides of rules 3.1, 4.1, and 4.2, which completely define $e2plus(s(0), y)$. 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 $e2plus(s(0), y) \rightarrow y + 2$ as the new lemma.

In order for the approach to be successful, a decidable theory $\mathcal{T}$ needs to have the following properties:

1. The no-theory condition introduced in [6] for $\mathcal{T}$ must be decidable, i.e., whether a term involving defined function symbols with $\mathcal{T}$-based definitions is equivalent to a $\mathcal{T}$-term (with any defined symbols) is decidable.
2. Given a finite set of equations relating parameterized terms in canonical form of $\mathcal{T}$, their solvability can be decided and furthermore, if solvable, find all possible most general solutions for parameters for all values of variables. This problem is related to the unification problem over $\mathcal{T}$. For example, in the case of the quantifier-free theory of Presburger arithmetic, 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 $k_1 = k_3 + k_3, k_4 = k_2, k_0 = 0$.
3. Terms in $\mathcal{T}$ have a small finite set of canonical forms (to keep the branching factor of the search space low); e.g., the quantifier-free theory of Presburger arithmetic has such a form $\Sigma k_i x_i + k_0$, where x_i are the variables and k_i are the parameters; similarly, the quantifier-free theory of free constructors admit also such canonical forms: a term in this theory is either a variable or one of $c_i(t_1, t_2)$, where c_i is a free constructor in the theory and t_1, t_2 are parametric terms.

The rest of the paper is organized as follows. Section 2 provides background and reviews the definition of a $\mathcal{T}$-based recursive definition, inductive validity, the cover set method, etc; more details can be found in [5, 6]. Section 3 introduces the procedure for generating simple lemmas. There are four major building blocks:

1. Given a conjecture of the form $f(s_1, \cdots, s_k) = r$, where r is a parametric term over $\mathcal{T}$, decide whether such an r can be found; if so, compute r. This operation is decidable for quantifier-free theories of free constructors and Presburger Arithmetic (PA); that is why the focus in this paper is on recursive function definitions over these two theories.

2. Given a finite set of parametric equations over $\mathcal{T}$, generate a finite representation of all assignment of parameters which make the equation true for all values of variables; further, if an assignment of parameters cannot be found for which the equation 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 equation is true.
3. Given a set of inconsistent parametric constraints associated with a given rule set, generate maximal consistent subsets of constraints and the associated rule sets.
4. Given a finite set of k-tuples of terms from a decidable theory $\mathcal{T}$, generate a preferably, more general and smaller set of k-tuples of terms that has the same set of ground instances as the input. This construction is needed to generate the left sides of conjectures. For a given $lhs = f(s_1, \cdots, s_k)$, where f has a $\mathcal{T}$-based definition and each s_i is a $\mathcal{T}$-term, compute from the definition of f, a complete rule set for normalizing all ground instances of lhs to a $\mathcal{T}$-term.

Section 4 is a brief discussion of how the procedure of Section 3 works for a quantifier-free theory of free constructors that may admit multiple canonical forms. Section 5 gives an overview of the notion of compatibility among recursive function definitions and shows how the proposed method can be used to generate lemmas with multiple defined symbols on their left side. This is illustrated using examples of lemmas needed in proofs of verification of generic arbitrary data-width and parametric arithmetic circuits; more details about these experiments can be found in [9, 10, 12].

2 Background

A brief overview of the relevant definitions and results from [11, 5] is given first; see also [6] where some of the results from [11] have been tightened. The example theories considered in this paper are the quantifier-free theory of Presburger arithmetic and the quantifier-free theory of free constructors (natural numbers which are generated by $0, s$ and finite lists which are generated by $nil, cons$ are two examples of this generic theory).

The framework of many-sorted first-order logic where "=" is the only predicate symbol is used below. For a set of function symbols $\mathcal{F}$ and an infinite set of variables $\mathcal{V}$, we denote the set of (well-typed) terms over $\mathcal{F}$ by $Terms(\mathcal{F}, \mathcal{V})$ and the set of ground terms by $Terms(\mathcal{F})$. Terms are represented as trees. Let $root(t)$ stand for the function symbol at the root of a term t and $\mathcal{V}(t)$ denote the variables of t. Given a theory $\mathcal{T}$, "$\models$" is the usual (semantic) first-order consequence relation; $\mathcal{F}_\mathcal{T}$ is the set of function symbols of $\mathcal{T}$; $Terms(\mathcal{F}_\mathcal{T}, \mathcal{V})$ denotes the terms of $\mathcal{T}$. Terms in $Terms(\mathcal{F}_\mathcal{T}, \mathcal{V})$ are called $\mathcal{T}$-terms. Below x^* stands for a k-tuple $\langle x_1, \cdots, x_k \rangle$.

Definition 1 ($\mathcal{T}$-based Function [11]). *A function $f \in \mathcal{F}$ is $\mathcal{T}$-*based *iff all rules $l \to r \in \mathcal{R}$ with $root(l) = f$ have the form $f(s^*) \to C[f(t_1^*), \ldots, f(t_n^*)]$, where $s^*, t_1^*, \ldots, t_n^*$ are k-tuples of $\mathcal{T}$-terms and C is a context over $\mathcal{F}_\mathcal{T}$. Moreover, we assume that all terms $f(t_i^*)$ are in normal form.*

Let D_f be the subset of rules $l \to r \in \mathcal{R}$ whose $root(l) = f$. D_f is called a $\mathcal{T}$-based definition of a $\mathcal{T}$-based function f. Henceforth, any $\mathcal{T}$-based definition is assumed to be terminating, sufficiently complete and ground confluent modulo $=_\mathcal{T}$, i.e., for every k-tuple of ground $\mathcal{T}$-terms g^*, $f(g^*)$ normalizes in finitely many steps to a unique ground term (equivalent modulo $=_\mathcal{T}$) in the range sort of f using D_f.

Definition 2 (Cover Set). *Given a $\mathcal{T}$-based function definition D_f of f, its* cover set *is $\mathcal{C}_f = \{\langle s^*, \{t_1^*, \ldots, t_n^*\}\rangle \mid f(s^*) \to C[f(t_1^*), \ldots, f(t_n^*)] \in \mathcal{D}_f\}$.*

Using the cover set method, an induction proof of a conjecture $q = r$ in which f appears, leads to the following subgoals for every $\langle s^*, \{t_1^*, \ldots, t_n^*\}\rangle \in \mathcal{C}_f$, the cover set of f:

$$[\sigma_1(q = r) \wedge \ldots \wedge \sigma_n(q = r)] \;\Rightarrow\; \theta(q = r), \tag{1}$$

where $\theta = \{x^* \to s^*\}$, and each $\sigma_i = \{x^* \to t_i^*\}$. If all formulas (1) are inductively valid, then by Noetherian induction, $q = r$ is also inductively valid.

Definition 3 (Simple Conjecture). *A conjecture $f(x_1, \cdots, x_k) = r$, where f is $\mathcal{T}$-based and $\langle x_1, \cdots, x_k\rangle$ are distinct variables, and r is a $\mathcal{T}$-term, is called simple.*

For example, the function *cost* above is based in Presburger arithmetic; the parameterized conjecture $P1$ about *cost* is a simple conjecture.

In [11], it is shown:

Theorem 1. *The inductive validity of a simple conjecture $f(x_1, \cdots, x_k) = r$ where f is a $\mathcal{T}$-based, can be decided based on its cover set.*

2.1 Quantifier-free Theory of Presburger Arithmetic (PA)

Following [6], we use the following definition for the *theory PA of Presburger Arithmetic*: $\mathcal{F}_{PA} = \{0, 1, +\}$ and AX_{PA} consists of the following formulas:

$$\begin{array}{rr} (x + y) + z = x + (y + z) & \neg\,(1 + x = 0) \\ x + y = y + x & x + y = x + z \Rightarrow y = z \\ 0 + y = y & x = 0 \vee \exists y.\ x = y + 1 \end{array}$$

For PA-term t with $\mathcal{V}(t) = \{x_1, \ldots, x_m\}$, there exist $a_i \in \mathbb{N}$ such that $t =_{PA} a_0 + a_1 \cdot x_1 + \ldots + a_m \cdot x_m$. Here, "$a \cdot x$" denotes the term $x + \ldots + x$ (a times) and "a_0" denotes $1 + \ldots + 1$ (a_0 times). For $s =_{PA} b_0 + b_1 \cdot x_1 + \ldots + b_m \cdot x_m$ and t as above, $s =_{PA} t$ iff $a_0 = b_0, \ldots, a_m = b_m$.

When we conjecture a term t involving symbols outside PA to be expressible in PA, i.e., equivalent to some term in PA but we do not know precisely which term, we have to use a *parametric* term of the form $\Sigma k_i x_i + k_0$, where $k_0, \cdots, k_n$ are unknown (hence, *parameters*). Let $\mathcal{K}$ be a finite set of parameters, distinct and disjoint from the variables $\mathcal{V}$. For t to be expressible in PA, there must exist values of $k_0, \cdots, k_n$ such that $t =_{PA} \Sigma k_i x_i + k_0$; otherwise, if t is not expressible in PA, then there cannot exist such values of $k_0, \cdots, k_n$.

There are thus two kinds of terms: (i) a PA-term consisting of variables and functions in PA, whose canonical form is $\Sigma a_i x_i + a_0$, where $x_i \in \mathcal{V}$ and $a_0, \cdots, a_m$ are nonnegative numbers, and (ii) a parametric term consisting of variables, parameters, and functions in PA, whose canonical form is $\Sigma a_i x_i + a_0$, where $x_i \in \mathcal{V}$ and $a_i, \cdots, a_m$ are linear polynomials in parameters in $\mathcal{K}$. Since the second kind of terms subsume the first kind, by a term in PA below, we mean the second kind of term and call it a *parametric* term, in contrast to the first kind of terms, which we will refer to as a *pure* term.

An (pure) equation in PA is $\Sigma a_i x_i + a_0 = \Sigma b_i x_i + b_0$, where $a_0, \cdots, a_m, b_0, \cdots, b_m$ are nonnegative numbers. Using cancellativity, we can simplify the equation so that every variable appears only on one side. A canonical form of the equation is $\Sigma c_i x_i + c_0 = 0$, where $c_0, \cdots, c_m$ are integers (including negative numbers); this is an abbreviation for the equation where variables with positive coefficients are on one side of the equation and variables with the negative coefficients are on the other side. For example, $2x - 3y - 1 = 0$ stands for $2x = 3y + 1$.

Since it is necessary to consider equations involving parametric terms as well, by a *parametric* equation, we mean $\Sigma p_i x_i + p_0 = 0$, where $x_i \in \mathcal{V}$ and each of $p_0, \cdots, p_m$ is a linear polynomial in parameters in $\mathcal{K}$ with integer coefficients. Again, since a parametric equation subsumes a pure equation, below we only refer to parametric equations.

A parametric equation is *valid* if and only if it is true for all values of its variables and parameters. As an example, the parametric equation $k_1\ x_1 + k_2\ x_2 = k_1\ x_1 + k_2\ x_2$ is valid.

A parametric equation is *strongly satisfiable* if and only if for all values of the variables, there exists parameter values that make equation true. As an example, the parametric equation $(k_1 + k_2)\ x_1\ +\ k_2\ x_2\ +\ k_3$ - $1 = 0$ is strongly satisfiable with parameter values $k_1 = k_2 = 0$ and $k_3 = 1$. Similarly, $(k_1 - k_2)\ x = 0$ is strongly satisfiable with parameter values $k_1 = k_2$. A parametric equation $\Sigma p_i x_i + p_0 = 0$ is strongly satisfiable if and only if there exist parameter values for which the conjunction of the equations $p_0 = 0, \cdots, p_m = 0$ is true.

A parametric equation is *weakly satisfiable* if and only if there exist variables and parameter values that make the equation true. As an example, the parametric equation $(k_1 + k_2)\ x_1 + k_2\ x_2 + k_1 - 1 = 0$ is weakly satisfiable for variable values $x_1 = x_2 = 0$ and parameter values $k_1 = 1$ (k_2 is indeterminate and can be any value). This equation is not strongly satisfiable since the conjunction of equations $k_1 + k_2 = 0$, $k_2 = 0$ and $k_1 - 1 = 0$ cannot be satisfied for any parameter values k_1 and k_2.

A parametric equation is *unsatisfiable* if there do not exist any values of parameters and variables which make the equation true. As an example, the equation $2k_1\ x$ - $1 = 0$ is unsatisfiable.

Every strongly satisfiable parametric equation is also weakly satisfiable. However, an equation may be weakly satisfiable under different sets of parameter values than the parameter values under which the equation is strongly satisfiable. As an example, consider the equation, $k_1 = k_1\ x\ +\ k_3$. The equation is strongly satisfiable when $k_1 = k_3 = 0$. However the equation is also weakly satisfiable for $x = 0$ and $k_1 = k_3 = 1$ (in fact any value insofar as $k_1 = k_3$).

A parametric equation that is weakly satisfiable but not strongly satisfiable is called *weak* parametric equation. Whether a given parametric equation $\Sigma p_i x_i + p_0 = 0$ is weak or not, can be easily checked by simplifying it to $p_0 = 0, \cdots, p_m = 0$ and finding a solution for parameters. If there is a solution, then the equation is not weak; otherwise it is weak.

A set S of parametric equations is *consistent* if and only if for all values of variables, there exist parameter values which make each equation in S true i.e., for all x_i there exists k_i such that the conjunction of the equations in the set is true. If S includes a weak parametric equation, it cannot be consistent. So a consistent S must only include strongly satisfiable parametric equations.

A *parametric constraint* is defined to be a linear equation over parameters. If a set S of parametric equations is consistent, then there exists a set of parametric constraints equivalent to it obtained by projecting (elimination of variables).

2.2 Quantifier-free Theory of Free Constructors

For the *theory $\mathcal{T}_C$ of free constructors*, $AX_{\mathcal{T}_C}$ consists of the following formulas.

$\neg c(x^*) = c'(y^*)$	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 $c \in \mathcal{F}_{\mathcal{T}_C}$
$\bigvee_{c \in \mathcal{F}_{\mathcal{T}_C}} \exists y^*.\ x = c(y^*)$	
$\neg\ (c_1(\ldots c_2(\ldots c_n(\ldots x \ldots) \ldots) \ldots) = x)$	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, "…" in the arguments of c_i stands for pairwise different variables.

A term in $\mathcal{T}_C$ can be either a variable or of the form $c_i(t_1, \cdots, t_k)$, where c_i is a constructor and $t_i's$ are terms in $\mathcal{T}_C$. As in PA, to check whether a term s involving symbols outside $\mathcal{T}_C$ is expressible in $\mathcal{T}_C$, we hypothesize s to be equivalent to some term in $\mathcal{T}_C$ without knowing exactly which one, i.e., s is equivalent to x, where $x \in s$ or $c_i(t_1, \cdots, t_k)$, where each t_i is a parameter standing for some term in $\mathcal{T}_C$ (expressed using variables in s).

A parametric term over $\mathcal{T}_C$ is thus a parameter, a variable, or $c_i(t_1, \cdots, t_k)$, where each t_i is a parametric term. Using the above axioms of $\mathcal{T}_C$, an equation over parametric terms can be checked for unsatisfiability. If an equation is satisfiable, then values of parameters making it true can be determined.

2.3 Splitting a Rule

In the procedure in Section 3, it becomes necessary to replace a rule in a definition by a finite set of *equivalent* rules. This splitting of a rule is guided by a substitution of variables in the rule. For a substitution σ assigning a variable x the term $\sigma(x)$, a finite set of terms including $\sigma(x)$ must be computed such that their ground instance cover all the ground terms in the sort of x.

Given a linear $\mathcal{T}$-term s in which variables appear at most once, let $cover(s)$ be a finite (preferably, minimal) set of terms in canonical forms including s such that every ground term of $sort(s)$ is equivalent in $\mathcal{T}$ to $\theta(t)$ for some ground substitution θ and $t \in cover(s)$. For examples, in the theory of free constructors, $cover(x) = \{x\}$ and $cover(c_i(x, y)) = \{c_0(\cdots), c_1(\cdots) \cdots c_i(x, y), \cdots, c_m(\cdots)\}$, where $c_0, \cdots, c_m$ are all the constructors of $sort(x)$. For PA, $cover(x) = \{x\}$; $cover(x + y) = \{x + y\}$, $cover(0) = \{0, x + 1\}$, $cover(n) = \{0, \cdots, n, x + n + 1\}$, etc.

Given a substitution $\sigma = \{x_1 \to s_1, \cdots, x_n \to s_n\}$, where s_i's are linear pair-wise disjoint (in variables) $\mathcal{T}$-terms in canonical form, the set of substitutions *complement* to σ, denoted as $compl(\sigma)$, is $\{\theta = \{x_1 \to t_1, \cdots, x_n \to t_n\} \mid \langle t_1, \cdots, t_n\rangle \in cover(s_1) \times cover(s_2) \times \cdots \times cover(s_n) \setminus \langle s_1, \cdots, s_n\rangle\}$[4].

Guided by a substitution σ, a rule $l \to r$ can be split into a finite equivalent set of rules $\{\theta(l) \to \theta(r) \mid \theta \in compl(\sigma) \cup \{\sigma\}\}$, denoted by $split(\{l \to r\}, \sigma)$. It is useful to normalize the right side of the rules (using the rule set under consideration).

Consider a substitution $\sigma = \{x \to 0, y \to 0\}$ over the theory of naturals generated by $0, s(x)$. Then $compl(\sigma) = \{\{x \to 0, y \to s(y')\}, \{x \to s(x'), y \to 0\}, \{x \to s(x'), y \to s(y')\}\}$ has three substitutions obtained by deleting the element $\{0, 0\}$ from the cross product $cover(0) \times cover(0)$. A rule such as $4 : e2plus(s(x), s(y)) \to s(e2plus(s(x), y))$ in section 1, can be split into four equivalent rules–one for each substitution in $\{\sigma\} \cup compl(\sigma)$. The rule $4.1 : e2plus(s(0), s(0)) \to s(e2plus(s(0), 0))$ is generated using σ; $4.2 : e2plus(s(0), s(s(y'))) \to s(e2plus(s(0), s(y')))$ comes from the first substitution in $compl(\sigma)$; two more rules are similarly generated from the other two substitutions in $compl(\sigma)$.

2.4 Abstracting Terms

In the procedure in Section 3, it will be necessary to generate a term from the left sides of a finite set of rules in a $\mathcal{T}$-based definition, in order to speculate an equational conjecture with that term as one of its sides. To perform this *abstraction* operation, we define from a given finite set L of k-tuples of variable-disjoint $\mathcal{T}$-terms, preferably a smaller set N of k-tuple of terms such that the ground instances of N and L are identical. (In the worst case, this operation will return L itself.)

The abstraction operation ABS on a finite set of tuples is defined recursively by first defining abs on a finite set of terms. For the theory of free constructors,

[4] Test sets used for checking the sufficient completeness property of a function definition can be used for this purpose; see, e.g., [7].

1. $abs(\{t\}) = \{t\}$ for any t;
2. $abs(\{x\} \cup L) = \{x\}$;
3. $abs(\{t_1, t_2\} \cup L) = abs(\{t_1\} \cup L)$ if t_2 is an instance of t_1;
4. $abs(\{c_1(x^*), c_2(y^*), \cdots, c_l(z^*)\} \cup L) = abs(\{x\} \cup L)$ where c_i's are all the l constructors of $\mathcal{T}_C$ and x does not appear in L;
5. $abs(\{c_i(s^*), \cdots, c_i(t^*)\} \cup L) = abs(\{c_i(u^*) \mid u^* \in ABS(\{s^*, \cdots, t^*\})\} \cup L)$ where L does not have any term with c_i as its outermost symbol.
6. otherwise, abs returns the input itself.

The function ABS on k-tuples is defined as: $ABS(\{s^*\}) = \{s^*\}$; $ABS(\{s_1^*, \cdots, s_m^*\} \cup L) = ABS(M \cup L)$, where $s_1^*, \cdots, s_m^*$ differ only in their i-th component,$1 \leq i \leq k$, and L does not include any k-tuple that is the same as s_1^* except for its i-th component, $M = \{t^* | t^* = s_1^*$ except for the i-th component and $t_i \in abs(\{s_{1i}, \cdots, s_{mi}\})\}$.

For PA, every ground term can be represented as 0 or $s^k(0), k > 0$, where s is a free constructor; every term with unique occurrences of variables can be represented as $x + y + \cdots + u$ or $s^k(x + y + \cdots + u)$. The function abs is defined as:

1. $abs(\{x + y + \cdots + u\} \cup L) = \{x\}$;
2. $abs(\{s^k(x + y + \cdots + u)\} \cup L) = \{s^k(x)\} \cup L$;
3. $abs(\{t_1, t_2\} \cup L) = abs(\{t_1\} \cup L)$ if t_2 is an instance of t_1;
4. $abs(\{0, s(x)\} \cup L) = abs(\{x\} \cup L)$, where x does not appear in L;
5. $abs(\{s(t_1), \cdots, s(t_m)\} \cup L) = abs(\{s(u) \mid u \in abs(\{t_1, \cdots, t_m\})\} \cup L)$ where L does not have any term with s as its outermost symbol;
6. otherwise, abs returns the input itself.

For instance, the three left sides $e2plus(s(0), 0)$, $e2plus(s(0), s(0))$ and $e2plus(s(0), s(s(x)))$ of rules 3.1, 4.1 and 4.2 in our running example can be used to formulate the left side of a new conjecture. To compute $ABS(\{\langle s(0), 0\rangle, \langle s(0), s(0)\rangle, \langle s(0), s(s(x))\rangle\})$, apply abs on the second component $\{0, s(0), s(s(x))\}$, which gives $abs(\{s(0), s(s(x))\} \cup \{0\}) = abs(\{s(x'), 0\})$ $= y$. The result of ABS is thus $ABS(\{\langle s(0), y\rangle\}) = \langle s(0), y\rangle$. The abstracted term is $e2plus(s(0), y)$, leading to the lemma $e2plus(s(0), y)) \rightarrow s(s(y)))$.

3 Generating Simple Lemmas

Before discussing the main procedure for generating simple lemma, the four main building blocks in the procedure are presented. The procedure can generate lemmas of the form $f(s_1, \cdots, s_k)$, where the subset of rules from D_f which can be used to normalize every ground instance of $f(s_1, \cdots, s_k)$ to a $\mathcal{T}$-term have left sides which match $f(s_1, \cdots, s_k)$; this condition is necessary to generate parametric constraints from the rules defining $f(s_1, \cdots, s_k)$ as shown below. Even though the presentation below is given using $\mathcal{T} = PA$, the procedure works for any decidable theory satisfying the properties discussed in the introduction. In the next section, the procedure is illustrated for $\mathcal{T}$ whose terms can have one of many canonical forms; in contrast, here terms of PA can be represented using a single canonical form.

3.1 Generating Right Side of a Conjecture

In the procedure, conjectures are speculated by constructing terms of the form $lhs = f(s_1, \cdots, s_k)$, where each s_i is a $\mathcal{T}$-term, and determining whether $f(s_1, \cdots, s_k)$ is equivalent to some $\mathcal{T}$-term, say rhs such that $lhs = rhs$ is an inductive consequence of D_f. For this, every possible candidate for rhs, i.e., every canonical form of a $\mathcal{T}$-term, is attempted. If no such rhs exists, then instances of lhs may be candidates to serve as the left side of other conjectures; this will be discussed in the next subsection. In subsection 3.3, a method for generating such lhs's is given; furthermore, a method for generating from the definition D_f of f, the rule set D_{lhs} consisting of rules necessary to normalize every ground instance of lhs to a $\mathcal{T}$-term is also presented.

For PA, $rhs = \Sigma k_i\ x_i + k_0$ and a solution to the parameters is attempted. The variables x_i's are all the variables in $\mathcal{V}(lhs)$. For each rule $l_i \rightarrow r_i$ in D_{lhs}, a parametric equation is generated by replacing each term of the form $f(t_1, \cdots, t_k)$ in the rule by $\sigma(\Sigma k_i x_i + k_0)$, where $\sigma(f(s_1, \cdots, s_k)) = f(t_1, \cdots, t_k)$[5]. 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 parametric equations so generated. A parametric equation generated from a rule may simplify using PA to a parametric constraint, which is a linear polynomial in parameters. If S includes an unsatisfiable parametric equation or a weak parametric equation, then S is not consistent. Values of parameters constrained by different parametric constraints may clash as well. If S is consistent, i.e., for all values of variables, parameter values can be found making each equation in S valid, then the conjecture $lhs = \delta(\Sigma k_i x_i + k_0)$, where δ is the most general solution of the set of parametric constraints generated from S, is a lemma. And, this implies that lhs is expressible in PA. The lemma generated is added to D_f to simplify other rules and lemmas.

In case S is inconsistent, parametric constraints in S are used to compute maximal consistent subsets from which instances of lhs that can serve as the left sides of new conjectures are generated as discussed in the next subsection.

3.2 Generating Maximal Consistent Subsets

In the previous step, if the set S of parametric equations generated form a given $lhs = f(s_1, \cdots, s_k)$ is inconsistent, S is used to generate instances of lhs possibly serving as the left sides of new conjectures about f. Toward this goal, every maximal consistent subset of S and the associated rule set are computed from S and the associated rule set D_{lhs}.

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 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 $c : p_0 = 0, \cdots, p_m = 0$ which can be included in S_{cn}, or it can be viewed as a weak parametric equation and be included in S_{wk}. For each $st \in S_{st}$, there are two cases leading to an exponentially many possibilities, each consisting of $\langle PC, WPE \rangle$, where PC is the set of parametric constraints and WPE is the set of weak parametric equations. For each maximal consistent subset MCS of PC, a most general solution for parameters δ is computed; let $R(MCS)$ stand for the associated rule set taken from D_{lhs}. The solution δ is then applied on each weak parametric equation in WPE to compute the values of variables, if any, such that the instantiated weak parametric equation is consistent with MCS. For any such substitution of variables of the weak parametric equation, the corresponding rule is split (see subsection 2.3) and $R(MCS)$ is extended to include the instance of the rule. The result is a rule set RF corresponding to MCS and instances of rules corresponding to weak parametric equations in WPE which are consistent with MCS. For each such possibility, the rule set RF is then used in the next subsection to speculate the left sides of the conjectures which are instances of lhs.

[5] Since nested recursive calls are not allowed in the right sides of rules in a PA-based function definition, a constraint so obtained is a parametric equation in PA.

For example, consider the inconsistent set of parametric equations $S = \{s1 : k_3 = 1, s2 : k_2 = 1, s3 : k_1\ x + k_3 = k_1, s4 : k_2 = 1\}$ generated from D_{e2plus} in section 1. $S_{cn} = \{s1 : k_3 = 1;\ s2 : k_2 = 1;\ s4 : k_1 = 1\}$, $S_{st} = \{k_1\ x + k_3 = k_1\}$, and $S_{wk} = \{\}$. There are two possibilities based on $\{k_1\ x + k_3 = k_1\}$: $\{\langle PC : \{s1 : k_3 = 1;\ s2 : k_2 = 1;\ s3 : k_1 = 0, k_3 = 0;\ s4 : k_2 = 1\}, WPE : \{\}\rangle, \langle PC : \{s1 : k_3 = 1;\ s2 : k_2 = 1;\ s4 : k_2 = 1\}, WPE : \{s3 : k_1\ x + k_3 = k_1\}\rangle\}$. There are two maximal consistent subsets $MCS_1 = \{k_1 = 0, k_3 = 0, k_2 = 1\}$ and $MCS_2 = \{k_3 = 1, k_2 = 1\}$.[6] Corresponding to MCS_1, the rule set is $\{2, 3, 4\}$. For MCS_2, an instance of rule 3 generated from $s3$ is computed giving x to be 0; the resulting rule set is $\{1, 2, 3.1, 4\}$.

3.3 Identifying Left Sides and their Complete Definitions

Given a subset R of rules of D_f which possibly generate a consistent set of parametric equations, left sides of conjectures need to be speculated. The rule set R is first preprocessed to ensure that most left sides of conjectures can be generated. After preprocessing, for every speculated left side $lhs = f(s_1, \cdots, s_k)$ 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 needs to be added to R to ensure this property (hopefully without violating the consistency condition).

Preprocessing Rules in R are refined by splitting based on unifying recursive calls with the left sides of nonrecursive rules in R. Let $l_i \rightarrow r_i$ be a non-recursive rule in R (i.e. r_i is a PA-term). For every recursive rule $l_j \rightarrow r_j$ in R, replace it by $split(\{l_j \rightarrow r_j\}, \sigma)$ if a recursive call in r_j unifies with l_i with the mgu σ. The expansion is performed for all possible pairs of non-recursive rules $l_i \rightarrow r_i$ and recursive rules $l_j \rightarrow r_j$ in R. The output of the preprocessing step is a rule set equivalent to the input rule set which may include instances of rules in the input rule set. We will abuse the notation and let R stand for the output of the preprocessing step.

Speculating Left sides The left sides of the rules in R are abstracted using ABS discussed in subsection 2.4 to generate k-tuples of terms to construct the left sides of new conjectures to be speculated. Let LHS be the finite set of possible left sides of the form $f(s_1, \cdots, s_k)$.

Generating a Complete Definition for a Left Side For each new left side $lhs = f(s_1, \cdots, s_k)$, its complete definition D_{lhs} as a set of rules is computed from R using the function *complete* which is initially invoked with $\{lhs\}$, R, $\{\}$ and a boolean flag set to $false$. The following invariant holds for each recursive call $complete(t, S_1, S_2, b)$; t is always of the form $f(s'_1, \cdots, s'_k)$, where each s'_i is a $\mathcal{T}$-term; every ground instance of t can be normalized to a $\mathcal{T}$-term using rules in $S_1 \cup S_2$.

1. $complete(\{t\} \cup L, R, D, b) = complete(L, R, D, b)$ if t rewrites using a rule in D.
2. $complete(\{t\} \cup L, R, D) = complete(\{\theta_j(t) \mid \theta_j \in compl(\sigma)\} \cup L, R, D, b)$ if t does not rewrite using a rule in D but there is a rule $l_i \rightarrow r_i$ in D such that $\sigma = mgu(t, l_i)$.
3. $complete(\{t\} \cup L, R, D, b) = complete(\{\theta_i(t) \mid \theta_i \in compl(\sigma)\} \cup L, R', D', false)$ where $l_i \rightarrow r_i$ is a rule in R such that $\sigma = mgu(t, l_i)$, $D' = D \cup \{\sigma(l_i) \rightarrow \sigma(r_i)\}$, and $R' = R$ - $\{l_i \rightarrow r_i\} \cup \{\theta_i(l_i) \rightarrow \theta_i(r_i) \mid \theta_i \in compl(\sigma)\}$.
4. $complete(\{\}, R, D, false) = complete(T, R, D, true)$, where $T = \{t_j \mid t_j$ appears as a recursive call to f in the right side of a rule in $D\}$.
5. $complete(\{\}, R, D, true) = D$.

[6] The reader might have noticed some duplication in computing maximal consistent subsets; better algorithms need to be investigated to avoid it.

For example, consider computing the complete definition of $lhs = f(0, s(y))$ with R being

1. $f(0,0) \rightarrow 0$,	2. $f(s(x),0) \rightarrow s(f(0,x))$,
3. $f(0,s(x)) \rightarrow s(f(x,0))$,	4. $f(s(x),s(y)) \rightarrow s(s(f(x,y)))$.

The initial call is $complete(\{f(0,s(y))\}, \{1,2,3,4\}, \{\}, false)$. The left side of rule 3 unifies with $f(0,s(y))$ by the substitution $x \rightarrow y$ for which the set of complement substitutions is empty; we thus have $complete(\{\}, \{1,2,4\}, \{3\}, false)$. By step 4, we get $complete(\{f(y,0)\}, \{1,2,4\}, \{3\}, true)$ based on the recursive call in rule 3. Since $f(y,0)$ unifies with rules 1, we get $complete(\{f(s(y),0)\}, \{2,4\},\{1,3\}, false)$; again using step 3 with rule 2, we get $complete(\{\}, \{4\}, \{1,2,3\}, false)$. From step 4, we get $complete(\{f(0,y), f(y,0)\}, \{4\}, \{1,2,3\}, true)$. Using step 2, this reduces to
$complete(\{f(0,s(y)), f(y,0)\}, \{4\}, \{1,2,3\}, true)$; using step 1, this reduces to:
$complete(\{f(y,0)\}, \{4\}, \{1,2,3\}, true)$. Again applying steps 2 and 1, respectively, we get $complete(\{\}, \{4\}, \{1,2,3\}, true)$, which terminates with $\{1,2,3\}$ as the rule set.

3.4 Procedure

The procedure for generating lemmas for PA given below is invoked with the most general $lhs = f(x_1, \cdots, x_m)$ and D_f, where $x_i' s$ are distinct variables. If a right side for this lhs can be found (implying that constraints generated from D_{lhs} are consistent), then lhs is expressible in PA and the procedure halts. Otherwise, rule sets associated with maximal consistent subsets of constraints are identified; for each such set, instances of lhs are speculated; for each such instance, say $\theta(lhs)$, a complete definition $D_{\theta(lhs)}$, sufficient to normalize every ground instance of $\theta(lhs)$ to a $\mathcal{T}$-term, is computed, and then the above steps are repeated.

To aid speculation of candidate lhs by abstracting left sides of rules in a maximal consistent set, the procedure expands the set by adding instances of rules from outside the set as well by preprocessing rules in the set. The rules generated are maintained along with each lhs in the form of two rule sets D_{lhs} the one completely defining lhs and the remaining rules $Rest_{lhs}$. On generation of a lemma L based on lhs, we consider all the rules associated with lhs along with L ($R = D_{lhs} \cup Rest_{lhs} \cup L$) and generate additional maximal consistent sets from this combined set of rules. This allows additional lemmas to be generated whose constraints are not consistent with L (and hence D_{lhs}). Generation of such lemmas may not be possible without considering the rules outside a maximal consistent set and regenerating maximal consistent sets since every maximal set originally generated may contain rules from D_{lhs}. The reader may refer to the Appendix on Ackermann's function, section 7, steps 5 and 6 for an example of such regeneration of maximal consistent sets.

For a given lhs, to ensure that parametric equations can be generated from each rule in D_{lhs}, the left side of each rule in D_{lhs} must match lhs using σ (i.e., $\sigma(lhs) = l$) and each recursive call t in r must either match lhs or $\sigma(t)$ must simplify to a $\mathcal{T}$-term using D_{lhs}.

```
Input: A $PA$-based, terminating, sufficiently-complete and ground-confluent definition $D_f$ of $f$.
Output: $\{f(s_1, \cdots, s_k) = r\}$ where each $s_i$ and $r$ are $PA$ terms such that
 $D_f \models_{\text{ind}} f(s_1, \cdots, s_k) = r$.
Method: Initially, $C_{lhs} = \{\langle f(x_1, \cdots, x_m), D_f, \{\}\rangle\}$.
 While there is an unmarked tuple $\langle lhs, D_{lhs}, Rest_{lhs}\rangle \in C_{lhs}$ do
  1. Find right side: Let $rhs$ be a parametric term in $PA$ expressed using variables in $lhs$.
    Generate a set of parametric equations $S$ using $D_{lhs}$ and the conjecture $lhs = rhs$.
    1.1 Success: If $S$ is consistent then let $\delta$ be the most general solution of $S$. Output lemma
        $\mathcal{L}$: $lhs = \delta(rhs)$. Mark $\langle lhs, D_{lhs}, Rest_{lhs}\rangle$ in $C_{lhs}$. Let $R = D_{lhs} \cup Rest_{lhs} \cup L$. Preprocess $R$.
        Let $S$ be the constraint set corresponding to the rules in $R$ after preprocessing.
        Go to step 2 for generating maximal consistent sets.
    1.2 Fail: $lhs = rhs$ is not expressible in $PA$; mark $\langle lhs, D_{lhs}\rangle$ in $C_{lhs}$. Go to step 2.
  2. Generate maximal consistent subsets of S: Generate from $S$, the set of pairs
      $\{\langle MCS, R(MCS)\rangle \mid MCS$ is a maximal consistent subset of S extended after instantiating
      weak parametric equations in $S\rangle\}$,where $R(MCS)$ is the rule set associated with $MCS$.
      Let $Rest_{lhs}$ be the other rules, that do not belong to $R(MCS)$.
  3. Identify left sides and definitions: From each pair $\langle MCS, R(MCS)\rangle$, identify new candidates
      for the left side of conjectures using $ABS$. For each new candidate $lhs$, generate a complete
      definition $D_{lhs}$ using the procedure complete. Add the unmarked tuple $\langle lhs, D_{lhs}$,
      $Rest_{lhs} \cup (R(MCS) - D_{lhs})\rangle$ to $C_{lhs}$.
      Repeat the above procedure.
```

4 Theories admitting Multiple Canonical Forms

The above procedure is an instance of a general procedure which works for decidable theories admitting multiple canonical form schemes. In the case of PA, a term in PA can be represented using a single parametric form. That is not true in general. A recursive data structure such as finite lists, has more than one canonical forms. In the theory of finite lists, a term can be a variable or *nil* representing an empty list, or *cons*(x, l), where x is an element and l is a list. Unlike PA, these canonical forms cannot be represented using a single scheme. The theory of finite lists is an instance of the generic theory of free constructors with m distinct constructors, say $c_1, c_2, \cdots, c_m, m > 1$.

Simple lemmas can be generated from recursive function definitions based in such theories by considering each canonical form scheme as the possible right side for the speculated left side of a conjecture. It is shown in [6] that as in the case of PA, no-theory condition for the quantifier-free theory of free constructors is decidable (i.e., whether a term $f(s_1, \cdots, s_k)$ is equivalent to a constructor term is decidable, where f is $\mathcal{T}$-based and each s_i is a constructor term). Furthermore, equations relating parametric terms in this theory can be solved for parameters as in PA. Below, we illustrate the general procedure with *append*:

$$1.\ append(nil, y) \rightarrow y, \qquad 2.\ append(cons(x_1, x), y) \rightarrow cons(x_1, append(x, y)).$$

Since finite lists admit as canonical form schemes, a variable, $nil, cons(t_1, t_2)$, where t_1, t_2 are themselves canonical form schemes, four conjectures with $append(x, y)$ as their left side are speculated with the right side being each of the canonical forms: nil, x, y, and $cons(t_1, t_2)$.

To check whether $append(x, y)$ can be equivalent to some constructor term s, nonrecursive rules are first used to generate constraints and perhaps rule out most of the candidate canonical forms. From the first rule in the definition, since there is a substitution σ such that $\sigma(append(x, y)) = append(nil, y)$, it must be the case that $\sigma(s) = y$. That is, when nil is substituted for x in s, it should be equivalent to y; since in the theory of free constructors, constructor terms cannot be simplified, $s = y$. It is easy to see that assuming $append(x, y) = y$, the constraint from the second rule, $y = cons(x_1, y)$, is unsatisfiable for any value of x_1, y. This implies that there is no constructor-term s such that $append(x, y) = s$. Speculating the right side to be nil, y, or $cons(t_1, t_2)$ can be shown not to lead to any lemmas being speculated either.

Conjecturing $append(x, y) = x$, however, results in the nonrecursive rule generating a constraint $nil = y$ which is weakly satisfiable only if $y = nil$. The second rule generates the constraint $cons(x_1, x) = cons(x_1, x)$, a valid formula. Combining the two constraints, a possible conjecture $append(x, nil) = x$ is speculated, which is valid and thus a lemma.

5 Generating Complex Lemmas

So far, the focus has been on generating *simple* lemmas of the form $f(s_1, \cdots, s_k) = r$, where r and each s_i are $\mathcal{T}$-terms. Using results from [11] (see also [6]), lemmas whose left sides include multiple occurrences of $\mathcal{T}$-based functions can also be generated. With this extension, it is possible to generate many nontrivial lemmas used in the verification of arithmetic circuits [9, 10, 12]. This is illustrated using an example in the next subsection.

To consider conjectures in which many $\mathcal{T}$-based function symbols can occur, it is first necessary to ensure that the definitions of function symbols appearing in such conjectures are *compatible* in the sense of [11, 6]. Compatibility ensure that when a term built from a compatible sequence of function symbols is instantiated and normalized, the result is a term in $\mathcal{T}$. Informally, g is compatible with f if for every rule of f, the *context* created by the rule on its right side can be simplified by the definition of g.

Definition 4 (Compatibility [6]). *Let g, f be $\mathcal{T}$-based, $f \notin \mathcal{F}_{\mathcal{T}}$, and $1 \leq j \leq m = arity(g)$. The definition of g* is compatible with the definition of f on argument j *iff for all rules α : $f(s^*, y^*) \to C[f(t_1^*, y^*), \ldots, f(t_n^*, y^*)]$, either $n = 0$ and $\alpha \in Exc_{g,f}$, or $g(x_1, \ldots, x_{j-1}, C[z_1, \ldots, z_n], x_{j+1}, \ldots, x_m) \to^*_{\mathcal{R}/\mathcal{T}}$ $D[\, g(x_1, \ldots, x_{j-1}, z_{i_1}, x_{j+1}, \ldots, x_m), \ldots, g(x_1, \ldots, x_{j-1}, z_{i_k}, x_{j+1}, \ldots, x_m) \,]$ 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.*

$Exc_{g,f}$ stands for the non-recursive rules in the definition of f whose right sides cannot be simplified by the definition of g [6]. Relaxing the requirement that g does not have to be compatible with the nonrecursive rules in the definition of f allows more function definitions to be compatible.

For example, $mod2$ and $half$ compute, respectively, the remainder and quotient on division by 2. These definitions are based in the theory of free constructors $0, s$ (and also PA).

1. $mod2(0) \to 0$,	2. $mod2(1) \to 1$,	3. $mod2(s(s(x))) \to mod2(x)$.
4. $half(0) \to 0$,	5. $half(1) \to 0$,	6. $half(s(s(x))) \to s(half(x))$.

It is easy to check that $half$ is compatible with $mod2$. However, $mod2$ is not compatible with $half$ since $mod2(half(s(s(x))))$ generated from rule 6 reduces to $mod2(s(half(x)))$ using the rule and cannot be simplified any further. However, $mod2$ is compatible with $mod2$, but $half$ is not compatible with $half$.

The compatibility property can be easily checked from the definitions of function symbols, and compatible sequence of function symbols can be easily generated.[7]

[7] 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 associative-commutative(AC) may also be used.

Using the procedure discussed in Section 3, it is easy to see that neither $mod2(x)$ nor $half(x)$ are expressible in PA. Furthermore, the procedure will not generate any simple lemma about $mod2$ and $half$.

Given PA-based function symbols g and f such that g is compatible with f, a conjecture $g(y_1, \cdots, f(x_1, \cdots, x_n), \cdots, y_k) = \Sigma k_i x_i + \Sigma k_j y_j + k_0$ 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 rest is the same as in Section 3. We illustrate the procedure using the example $half(mod2(x))$.

Given $half(mod2(x)) = k_1\ x + k_2$, 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 $half(mod2(0)) = k_1\ 0 + k_2$, which simplifies using rules 1 and 4 to the constraint $k_2 = 0$. From rule 2, x is instantiated to 1, giving $half(mod2(1)) = k_1\ 1 + k_2$, which simplifies using rules 2 and 5 to the constraint $k_1 + k_2 = 0$. From rule 3, x is instantiated to $s(s(u))$ giving the conclusion subgoal $half(mod2(s(s(u)))) = k_1\ u + k_1 + k_1 + k_2$ assuming the induction hypothesis $half(mod2(u)) = k_1\ u + k_2$, obtained by instantiating x to be u. After simplifying using rules 3 and 6, 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. And, lemma $half(mod2(x)) = 0$ is generated.

The notion of compatibility can be extended to that of a *compatible sequence* involving function symbols $[f_1, \cdots, f_m]$ where each f_i is compatible with f_{i+1}, $1 \leq i \leq m - 1$. Given a sequence $[g, f_1, f_2]$, for example, we can generate the left side of conjectures of the form $g(y_1, \cdots, f_1(x_1, \cdots, f_2(z_1, \cdots, z_m), \cdots, x_n) \cdots, y_k)$ such that the variables $x_i's$, $y_j's$ and $z_k's$ are distinct and the variables $z_k's$ the inductive positions of f_2[8] do not appear in $x_i's$ and $y_j's$. The left side of a conjecture thus generated can be checked for the no-theory condition.

If a conjecture generated from a compatible sequence of function definitions cannot be expressed in $\mathcal{T}$, then appropriate instances of the conjecture are found similar to Section 3. Some of the lemmas generated by the proposed approach based on the theories of PA and finite lists are given in Figure 1. Automatic generation of one such lemma is illustrated in the next subsection, where some of the function appearing in Figure 1 are also defined.

Lemma	**Theory**	**Remarks**
$bton(radder(ntob(x), ntob(x), ntob(x))) = x + x + x$	Finite lists, PA	Correctness of ripple carry
$bton(cadder(ntob(x), ntob(x), ntob(x))) = x + x + x$	Finite lists, PA	Correctness of carry save
$bton(leftshift(ntob(x))) = x + x$	Finite lists	Correctness of left-shift
$bton(pad0(ntob(x))) = x$	Finite lists, PA	Correctness of Padding
$compl(compl(x)) = x$	Finite lists	Complement's idempotence
$half(mod2(x)) = 0$	PA	Multiple functions
$mod2(mod2(x)) = 0$	PA	Multiple functions
$ack(1, n) = n + 2$	PA	Nested recursive calls
$ack(2, n) = 2n$ - 3	PA	Nested recursive calls
$rev(rev(x)) = x$	Finite lists	Multiple canonical forms
$append(x, nil) = x$	Finite lists	Multiple canonical forms

Fig. 1. Some Examples of Generated Lemmas

[8] positions of f_2 involved in recursion in the definition of f_2.

5.1 Generating Lemmas for Arithmetic Circuits

Consider the following definitions of *bton*, *ntob* and *pad0*. The functions *bton* and *ntob* convert binary to decimal representations and vice versa, respectively. The function *pad0* adds a leading binary zero to a bit list. Bits increase in significance in the list with the first element of the list being the least significant. These functions are used to reason about number-theoretic properties of parameterized arithmetic circuits [8, 10].

1. $bton(nil) \rightarrow 0$, 2. $bton(cons(b0, y_1)) \rightarrow bton(y_1) + bton(y_1)$,
3. $bton(cons(b1, y_1)) \rightarrow s(bton(y_1) + bton(y_1))$.
4. $ntob(0) \rightarrow cons(b0, nil)$, 5. $ntob(s(0)) \rightarrow cons(b1, nil)$,
6. $ntob(s(s(x_2 + x_2))) \rightarrow cons(b0, ntob(s(x_2)))$,
7. $ntob(s(s(s((x_2 + x_2))))) \rightarrow cons(b1, ntob(s(x_2)))$
8. $pad0(nil) \rightarrow cons(b0, nil)$, 9. $pad0(cons(b0, y)) \rightarrow cons(b0, pad0(y))$,
10. $pad0(cons(b1, y)) \rightarrow cons(b1, pad0(y))$.

The underlying decidable theory $\mathcal{T}$ is the combination of the quantifier-free theories of finite lists of bits and PA; $b0$, $b1$ stand for binary 0 and 1. Definitions of *bton*, *ntob*, and *pad0* are $\mathcal{T}$-based. The function *pad0* is compatible with *ntob*; *bton* is compatible with *pad0*. Hence $[bton, pad0, ntob]$ form a compatible sequence.

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 lemma needed to establish the correctness of such circuits is that the padding does not affect the number output by the circuit. We illustrate how this lemma can be automatically generated.

A conjecture with $bton(pad0(ntob(x)))$ as the left side is attempted. Since the sort of *bton* is natural numbers, the right side of the conjecture is $k_1\ x + k_0$. As before, parametric equations are generated corresponding to the instantiations of x for each of the four rules defining the innermost function *ntob*.

- From rule 4, x gets instantiated to 0, giving $bton(pad0(ntob(0))) = k_1\ 0 + k_2$. This simplifies using the rules 1, 9, 8, 2, and 1 and PA to the constraint $s1$: $0 = k_2$.
- From rule 5, x gets instantiated to 1, giving $bton(pad0(ntob(1))) = k_1\ 1 + k_2$. This simplifies using the rules 5, 10, 8, 2, and 1 and PA to the constraint $s2$: $1 = k_1 + k_2$.
- From rule 6, x is instantiated as $s(s(x_2+x_2))$ for the conclusion subgoal with x instantiated as $s(x_2)$ for the induction hypothesis. This gives $bton(pad0(ntob(s(s(x_2+x_2))))) = k_1\ s(s(x_2+x_2)) + k_2$ as the conclusion with $bton(pad0(ntob(s(x_2)))) = k_1 s(x_2) + k_2$ as the hypothesis. The conclusion simplifies using rules 6, 9, 2 to $bton(pad0(ntob(s(x_2)))) + bton(pad0(ntob(s(x_2)))) = k_1\ s(s(x_2+x_2)) + k_2$. Applying the hypothesis gives $k_1\ s(x_2) + k_2 + k_1\ s(x_2) + k_2 = k_1\ s(s(x_2 + x_2)) + k_2$, which gives the constraint $s3$: $k_2 = 0$.
- Parametric equation from rule 7 is similarly generated and gives constraint $s4$:$k_2 = 0$.

Taking the conjunction of all the constraints gives $\{k_1 = 1, k_2 = 0\}$, implying that it is a consistent set. This gives the lemma

$$bton(pad0(ntob(x))) = x,$$

which states that $pad0$ does not change the numeric representation of a bit list as required in the verification of properties of multiplier circuits. in [8, 12].

6 Conclusion and Further Research

A procedure for automatically generating lemmas from recursive function definition is given using decision procedures. It is expected that after a user has written recursive definitions of functions on theories such as PA and on constructors, the procedure will automatically start generating lemmas whose one side has terms from the underlying decidable theory. In this way, a library of useful lemmas can be automatically built with the hope that these lemmas will be found useful in the proofs of other nontrivial lemmas. This is demonstrated using examples about commonly used function definitions on numbers and finite lists, as well as from our experiments in automatically verifying number-theoretic properties of arithmetic hardware circuits.

Conjectures considered in this paper have one side to be a term from a decidable theory. In [5], the approach proposed in [11] has been extended to decide quantifier-free formulas which are boolean combinations of such equations. Recently in [6], a decision procedure for a class of equations in which function symbols with $\mathcal{T}$-based definitions can appear on both sides of equations has been developed. The proposed procedure needs to be extended so that a wider class of lemmas can be automatically generated from $\mathcal{T}$-based function definitions.

We have discussed two well-known quantifier-free decidable theories on numbers–the theory of free constructors $0, s$, and the quantifier-free theory of Presburger arithmetic. Most of the examples in the paper are done in the framework of the quantifier-free theory of Presburger arithmetic and the theory of finite lists. There exists even a more expressive and decidable theory of numbers involving $0, 1, s, +, 2^y, \geq$; we will call it the theory of *2exp*. These three theories define a strict hierarchy in their expressive power about properties of numbers. Given a function definition on numbers based in a theory $\mathcal{T}$, it is possible to speculate conjectures using the most expressive theory (e.g., the theory of $2exp$). If a given conjecture cannot be expressed in the most expressive theory in the hierarchy, then it cannot be expressed in any theory in the hierarchy. However, the most expressive the theory, solving parametric equations in it can be more complex. There is thus a trade-off. An alternate method is to start with the least expressive power (e.g., $\mathcal{T}$, the theory in which the definition is based), and then move to more expressive theories for conjectures which cannot be expressed in $\mathcal{T}$. The proposed approach will work with either of the two heuristics for trying theories in different orders. In fact, some of the examples in the paper, e.g. $e2plus$ could have been attempted using the theory of exponentiation. The function $e2plus$, for instance, can be shown to be expressible in this theory. Similarly, instances of Ackermann's function including $ack(0, y), ack(1, y), ack(2, y), ack(3, y)$ can all be expressed in the theory of exponentiation. The expanded version, available from `http://faculty.ist.unomaha.edu/msubramaniam/subuweb.htm`, includes an Appendix in which derivation of lemmas about Ackermann's function is discussed. Ackermann's function is not even PA-based, because of nested recursive calls in it; however, the proposed approach still works on it provided parametric equations in PA are extended to include non-linear polynomials over parameters as coefficients of variables as well as limited reasoning about exponentiation and multiplication are available to simplify parametric constraints.

Acknowledgment: We thank Jürgen Giesl for collaborative work which triggered some of the ideas in this paper.

References

1. F. Baader & T. Nipkow. *Term Rewriting and All That.* Cambridge Univ. Pr., 1998.
2. R. S. Boyer and J Moore. *A Computational Logic.* Academic Press, 1979.
3. A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, & A. Smaill. Rippling: A Heuristic for Guiding Inductive Proofs. *Artificial Intelligence*, 62:185-253, 1993.
4. H. B. Enderton. *A Mathematical Introduction to Logic.* 2nd edition, Harcourt/ Academic Press, 2001.
5. J. Giesl & D. Kapur. Decidable Classes of Inductive Theorems. *Proc. IJCAR '01*, LNAI 2083, 469-484, 2001.

6. J. Giesl & D. Kapur. Deciding Inductive Validity of Equations. *Proc. CADE '03*, LNAI 2741, Miami, 2003. An expanded version appeared as Technical Report AIB-2003-03, 2003 is available from `http://aib.informatik.rwth-aachen.de`.
7. D. Kapur, P. Narendran and H. Zhang. Automating Inductionless Induction using Test Sets. *Journal of Symbolic Computation,* 11 (1-2), 81-111, 1991.
8. D. Kapur & M. Subramaniam. New Uses of Linear Arithmetic in Automated Theorem Proving by Induction. *Journal of Automated Reasoning*, 16:39–78, 1996.
9. D. Kapur and M. Subramaniam. Mechanically verifying a family of multiplier circuits. *Proc. Computer Aided Verification (CAV),* Springer LNCS 1102, 135-146, August 1996.
10. D. Kapur and M. Subramaniam. Mechanical verification of adder circuits using powerlists. *Journal of Formal Methods in System Design,* 13 (2), page 127-158, Sep. 1998.
11. D. Kapur & M. Subramaniam. Extending Decision Procedures with Induction Schemes. *Proc. CADE-17*, LNAI 1831, 324-345, 2000.
12. D. Kapur and M. Subramaniam. Using an induction prover for verifying arithmetic circuits. *Int. Journal of Software Tools for Technology Transfer,* Springer Verlag, 3(1), 32-65, Sep. 2000.
13. D. Kapur & H. Zhang. An Overview of Rewrite Rule Laboratory (*RRL*). *Journal of Computer and Mathematics with Applications*, 29:91–114, 1995.
14. R. E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1-12, 1984.
15. C. Walther. Mathematical Induction. Gabbay, Hogger, & Robinson (eds.), *Handbook of Logic in Artificial Intelligence and Logic Programming, Vol. 2*, Oxford University Press, 1994.
16. W.A. Hunt, R. B. Krug & J. Moore, Linear and Non-linear Arithmetic in ACL2, *Proc. CHARME03*, D. Geist (ed.), LNCS 2860, 2003.
17. H. Zhang, D. Kapur, & M. S. Krishnamoorthy. A Mechanizable Induction Principle for Equational Specifications. *Proc. CADE-9*, LNCS 310, 1988.

7 Appendix: Ackermann's Function - a non-trivial Example

Consider the following recursive definition D_a of Ackermann's function over natural numbers. The right side of the third rule has nested recursive calls. The definition of a is thus not $\mathcal{T}$-based, where $\mathcal{T}$ is the theory of free constructors $0, s$ (or even PA). The definition 2 of a $\mathcal{T}$-based function can be generalized to allow nested recursive calls in the right side of a rule. Recursive calls appearing in r of a rule can be repeatedly abstracted by variables from inside-out, and the result of this repeated abstractions is required to be a $\mathcal{T}$-term. For example, in the third rule below, if the innermost recursive call to a in the right side is abstracted, then the right side becomes $a(x, w)$, which is abstracted to a variable.

The above procedure works even on definitions such as $a(x, y)$, resulting in useful lemmas. Because of nested recursive calls, constraints generated fall outside PA as will be illustrated below. The constraint is still linear in variables, but coefficients of the variables can be nonlinear polynomials in parameters. Different machinery will have to be employed in solving such parametric equations.[9]

$$\begin{array}{ll} 1. & a(0, y) \rightarrow s(y), \\ 2. & a(s(x), 0) \rightarrow a(x, s(0)), \\ 3. & a(s(x), s(y)) \rightarrow a(x, a(s(x), y)). \end{array}$$

1. *Find Right Side for $a(x, y)$:* Let $lhs = a(x, y)$, $R = D_a$. The conjecture using the canonical form for PA is: $a(x, y) = k_1\ x + k_2\ y + k_3$.
   To solve for the parameters k_1, k_2, and k_3, three constraints are generated from the rules in D_a. The first constraint $s1 : k_2\ y + k_3 = y + 1$ comes from the first rule. It is generated by replacing the left side of the rule by $k_1\ 0 + k_2\ y + k_3$. The second constraint is generated from the second rule in a similar fashion and it simplifies to $s2 : k_1 = k_2$. The third constraint simplifies to $s3 : k_1 + k_2\ y + k_2 = k_2\ k_1\ x + k_2\ k_1 + k_2\ k_2\ y + k_2\ k_3$. Substitutions are made for each of $a(s(x), s(y))$, $a(s(x), y)$, $a(x, a(s(x), y))$. The reader will notice that the resulting constraint is linear in variables x, y, but nonlinear in parameters k_1, k_2, k_3 because of nested recursive calls.
   These constraints must be simultaneously satisfied for all possible values of x, y. Constraints $s1$, $s2$ restrict $k_1 = k_2 = k_3 = 1$, whereas $k_1 k_2$, the coefficient of x in $s3$ must be 0, thus implying that these constraints cannot be satisfied for all values of x. Thus, a is not expressible in PA.
   We now attempt to find a proper instance $a(s_1, s_2)$ of $a(x, y)$ which could serve as the left side of a lemma to be speculated.
2. *Generate Maximal Consistent Subsets:* The constraint set $MCS = \{s1, s2\}$ constitutes a maximal consistent subset with the most general solution, $\delta = \{k_1 = k_2 = k_3 = 1\}$. The set is expanded by adding specific instances of the constraint $s3$ obtained by instantiating its variables. To determine the values of the variables x and y in $s3$, the solution δ is applied to $s3$, which gives $x + y = y$ that simplifies to $x = 0$. Using this value of x, rule 3 splits into

   $$\begin{array}{ll} 3.1 & a(s(0), s(y)) \rightarrow s(a(s(0), y)), \\ 3.2 & a(s(s(x)), s(y)) \rightarrow a(s(x), a(s(s(x)), y)). \end{array}$$

   The above two rules 3.1 and 3.2 are maintained along with the rule 3 with the overall set of rules $R = \{1, 2, 3, 3.1., 3.2\}$. Rule 3.1 is added to MCS with constraint $s3.1$: $k_2 = 1$.
   The maximal consistent is further preprocessed by unifying the recursive call in rule 2 with the left side of rule 1. This splits rule 2 into:

[9] The following property of $*$, $k_1 k_2 = 0 \Rightarrow (k_1 = 0 \vee k_2 = 0)$, turns out to be effective and suffices in many cases. This property is already used in theorem provers such as RRL and ACL2 [16] to do limited reasoning about $*$.

2.1 $a(s(0), 0) \rightarrow 2,$
2.2 $a(s(s(x)), 0) \rightarrow a(x, a(x, s(0))).$

The constraint for rule 2.1 is $s2.1$: $k_1 + k_3 = 2$, which is consistent with MCS. Hence MCS is expanded by adding rule 2.1. The constraint for rule 2.2, $s2.2$: 2 $k_1 = k_2k_1\ x + k_2k_2 + k_2k_3$ is not solvable and hence is not added to MCS.[10]. $R(MCS) = \{1, 2.1, 3.1\}$ and $Rest_{lhs} = \{2.2, 3.2\}$. The set of rules R after this step contains $R = \{1, 2, 3, 3.1, 3.2, 2.1, 2.2\}$.

3. *Identify new candidate lhs*: A candidate left side obtained by abstracting left sides of rules, 2.1 and 3.1 in $R(MCS)$ is $a(1, m)$.
4. *Compute D_{lhs}*: Using the function *complete* to determine a complete definition for $a(1, m)$ based on MCS produces $D_{a(1,m)} = \{2.1, 3.1\}$. Note that $D_{a(1,m)}$ satisfies the required conditions that every left side of a rule as well as the recursive calls in $D_{a(1,m)}$ match $a(1, m)$. Hence parametric equations can be generated from each rule in $D_{a(1,m)}$ to find right side of the conjecture as shown in the next step. The set of rules $Rest_{a(1,m)} = \{1, 2.2, 3.2\}$. At the end of this step, the candidate set C_{lhs} is updated with the tuple $\langle a(1, m), \{2.1, 3.1\}, \{1, 2, 2.2, 3, 3.2\}\rangle$.
5. *Find Right Side for* $a(1, m)$: The constraints for rules 2.1 and 3.1 are, respectively, $s2.1 : k_1 + k_3 = 2$ and $s3.1 : k_2 = 1$ and they are solvable, implying that $a(1, y)$ is expressible in PA. Applying the most general solution $\delta = \{k_1 + k_3 = 2, k_2 = 1\}$ to $a(1, y) = k1 + k2y + k3 = 2 + y$, the lemma generated is,

$$4 : a(1, y) = s(s(y))$$

The constraint associated with the lemma is $s4 : k_1 + k_2y + k_3 = y + 2$. Rule 4 is added to the rule set $R = \{1, 2, 2.1, 2.2, 3, 3.1, 3.2\}$ and the rules are pre-processed.
This set can be further expanded by pre-processing by splitting rule 2.2 based on substitution $\{x \rightarrow 0\}$ (obtained by unifying the recursive call on its right side with the left side of rule 4) into

2.2.1 $a(2, 0) \rightarrow 3,$
2.2.2 $a(s(s(s(x))), 0) \rightarrow a(s(x), a(s(x), s(0)))$

Similarly,(by unifying its outer recursive call with left side of rule 4) rule 3.2 is split into:

3.2.1 $a(2, s(y)) \rightarrow s(s(a(2, y))),$
3.2.2 $a(s(s(s(x))), s(y)) \rightarrow a(s(s(x)), a(s(s(s(x))), y)).$

The rule set R after pre-processing is $\{1, 2, 2.1, 2.2, 2.2.1, 2.2.2, 3, 3.1, 3.2, 3.2.1, 3.2.2, 4\}$.
6. *Generate Maximal Consistent Sets:* One of the maximal consistent sets obtained from the rule set R is $R(MCS) = \{2.1, 2.2.1, 3.2.1\}$. The rest of the rules $Rest_{lhs} = \{1, 2, 2.2.2, 3, 3.2.2, 4\}$.
7. *Identify Candidate lhs*: The left side of rules 2.2.1 and 3.2.1 can be abstracted to $a(s(s(0)), m)$. Thus, $lhs = a(2, m)$.
8. *Compute D_{lhs}*: Using the function *complete* to generate a complete definition of $a(2, m)$ gives $D_{a(2,m)} = \{2.2.1, 3.2.1\}$. Note that $D_{a(2,m)}$ satisfies the required conditions that every left side as well as the recursive calls in the rules in $D_{a(2,m)}$ match with $a(2, m)$, parametric equations can be generated using $D_{a(2,m)}$ to find the right side of $a(2, m)$ as shown in the next step. The remaining rules $Rest_{a(2,m)} = \{1, 2, 2.1, 2.2, 2.2.2, 3, 3.1, 3.2, 3.2.2, 4\}$.
At the end of this step, the candidate set C_{lhs} is updated with the tuple $\langle a(2, m), D_{a(2,m)}, Rest_{a(2,m)}\rangle$.

[10] The set MCS can be expanded further by instantiating 2.2 further with $x = 0$ as discussed below.

9. *Find Right Side for* $a(2, m)$: The constraints $s2.2.1 : 2k_1 + k_3 = 3$ and $s3.2.1 : k_2 = 2$ can be solved, implying that $a(2, y)$ is expressible in PA. Applying the most general solution, to $\delta = \{2k_1 + k_3 = 3, k_2 = 2\}$ to $a(2, m) = 2k_1 + k_2\ m + k_3$, the lemma generated is
$$5 : a(2, y) = s(s(s(y + y))).$$
The constraint associated with this lemma is $s5 : 2k_1 + k_2\ y + k_3 = 2y + 3$. Lemma 5 is added to the rule set $\{1, 2, 2.1, 2.2, 2.2.1, 2.2.2, 3, 3.1, 3.2, 3.2.1, 3.2.2, 4\}$. Rules are pre-processed and maximal consistent subsets are generated and the procedure is repeated.
10. No additional lemmas are generated from the rule set in the above step since $a(3, y)$ and $a(x, 0)$ are both not expressible in PA, and Ackermann's function is monotonic in both arguments.

7.1 Generating Lemmas Using $2exp$ Theory

As mentioned earlier, there exists a more expressive theory than PA, the theory $2exp$ with functions 0, 1, +, 2^y. The canonical form of the theory $2exp$ is: $2^{\Sigma k_i\ x_i + k_0} + \Sigma l_i\ x_i + l_o$ for *integer* parameters $k_i's$ and $l_j's$. Instead of PA it can be checked whether the Ackermann's function a is expressible in $2exp$ by formulating the conjecture: $a(x, y) = 2^{k_1\ x + k_2\ y + k_0} + l_1\ x + l_2\ y + l_0$. By generating parametric equations from the rules 1-3 in D_a it can be determined that there is no value of parameters that make these equations consistent. Hence a is not expressible in the theory of $2exp$ as well.

A similar procedure as done for PA above can be followed to attempt to generate lemmas with left sides more specific than $a(x, y)$. The two lemmas $a(1, y) \rightarrow s(s(y))$ and $a(2, y) \rightarrow s(s(s(y + y)))$ can be generated in this case as well. However, using $2exp$ we can generate an additional lemma, $a(3, y) \rightarrow 2^{y+3}$ - 3 whose right side is not expressible in PA.

We describe below how these lemmas can be generated using the theory of $2exp$.

1. *Find Right Side for* $a(x, y)$*:* To determine if $a(x, y)$ is expressible in $2exp$ three sets of constraints are generated using the 3 rules in D_a. The parametric equation generated from the first rule is
$$s(y) = 2^{k_1\ 0 + k_2\ y + k_0} + l_1\ 0 + l_2\ y + l_0, \tag{2}$$
which simplifies to the constraint $s1 : \{k_0 = k_2 = 0, l_0 = 0, l_2 = 1\}$.
The second rule leads to the parametric equation
$$2^{k_1\ x + k_1 + k_2\ 0 + k_0} + l_1\ x + l_1 + l_2\ 0 + l_0 = 2^{k_1\ x + k_2\ 1 + k_0} + l_1\ x + l_2\ 1 + l_0, \tag{3}$$
which simplifies to the constraint $s2 : \{k_1 = k_2, l_1 = l_2\}$.
The parametric equation generated from the third rule is
$$2^{k_1\ x + k_1 + k_2\ y + k_2 + k_0} + l_1\ x + l_1 + l_2\ y + l_2 + l_0 = 2^{k_1\ x + k_2\ A + k_0} + l_1\ x + l_2\ A + l_0, \tag{4}$$
where $A = 2^{k_1\ x + k_1 + k_2\ y + k_0} + l_1\ x + l_1 + l_2\ y + l_0$. By simplifying the parametric equation (by equating the corresponding linear expressions in the exponent of 2 and the corresponding linear expressions involving the parameters $l_j's - k_1\ x + k_1 + k_2\ y + k_2 + k_0 = k_1\ x + k_2\ A + k_0$; $l_1\ x + l_1 + l_2\ y + l_2 + l_0 = l_1\ x + l_2\ A + l_0$, we get the third constraint $s3$: $\{k_1 = 0 = k_2, l_1 = l_2 = 0\}$.

2. *Generate Maximal Consistent Sets:* The constraint set $S = \{s1, s2, s3\}$ is not consistent since the constraint $s3$ restricts l_2 to 0 which contradicts $s1$. A maximal consistent subset is $MCS = \{s1, s2\}$, with the most general solution $\delta = \{k_0 = k_1 = k_2 = 0, l_0 = 0, l_1 = l_2 = 1\}$. Applying δ to (4), we get $2^0+x+1+y+1+0 = 2^0+x+1\ (2^0+x+1+y+0)+0$, which gives the specific value of x to be 0. Rule 3 can be split using $x = 0$ as before into rules 3.1 and 3.2, given in the previous subsection. The rule 3.1 is added to MCS with constraint $s3.1$: $\{k_2 = 0, l_1 = 1\}$.
   As done in the case of PA the rules can be further pre-processed by unifying the recursive call in rule 2 with the left side of rule 1 to produce rules 2.1 and 2.2. given in the previous subsection. The parametric equation for rule 2.1 is
   $$2^{k_1\ 1+k_2\ 0+k_0} + l_1\ 1 + l_2\ 0 + l_0 = 2, \tag{5}$$
   which gives the constraint $s2.1$: $\{k_1 = k_2 = k_0 = 0, l_1 + l_0 = 1\}$ which is consistent with MCS. Hence rule 2.1 is added to MCS. As seen in the previous section, rule 2.2 is found to be inconsistent and hence is not added to MCS.
3. *Identify new candidate lhs*: A candidate $lhs = a(1, y)$ is obtained by abstracting the left sides of rules 2.1 and rule 3.1.
4. *Compute* D_{lhs}: Rules 2.1 and 3.1 form a complete definition of the $a(1, y)$ as before.
5. *Find Right Side for* $a(1,m)$: Rule 4, $a(1, y) \rightarrow s(s(y))$ is generated as a lemma from the constraints $s2.1$ and $s3.1$ in a similar fashion as done for PA.
6. *Find Right Side for* $a(2,m)$: Adding the rule 4 to the rule set $\{1, 2, 2.1, 2.2, 3, 3.1, 3.2\}$ and repeating the process as done for PA similarly leads to the other lemma $a(2, y) \rightarrow s(s(s(y+y)))$, rule 5 above. This lemma is generated using the theory of $2exp$ by splitting the rule 2.2 into 2.2.1 and 2.2.2 and the rule 3.2 into rules 3.2.1 and 3.2.2 as described in the case of PA.
   The lemma 5 can be added to the expanded rule set, rules are preprocessed and maximal consistent sets are regenerated.
   Pre-processing splits rule 2.2.2 into two rules.
   $$\begin{array}{l} 2.2.2.1 : a(3, 0) \rightarrow 5 \\ 2.2.2.2 : a(s(s(s(s(x)))), 0) \rightarrow a(s(s(x)), a(s(x), a(x, a(x, s(0))))). \end{array}$$
   Similarly, rule 3.2.2. is split into rules
   $$\begin{array}{l} 3.2.2.1 : a(3, s(y)) \rightarrow s(s(s(a(3, y) + a(3, y)))) \\ 3.2.2.2 : a(s(s(s(s(x)))), s(y)) \rightarrow a(s(s(s(x))), a(s(s(s(s(x)))), y)). \end{array}$$
7. *Generate Maximal Consistent Subsets:* The parametric equation for rule 2.2.2.1 is
   $$2^{k_1\ 3+k_2\ 0+k_0} + l_1\ 3 + l_2\ 0 + l_0 = 5. \tag{6}$$
   The parametric equation for rule 3.2.2.1 is
   $$2^{k_1\ 3+k_2\ y+k_2+k_0} + l_1\ 3 + l_2\ y + l_2 + l_0 = 2^{k_1\ 3+k_2\ y+k_2+k_0+1} + l_1\ 6 + l_2\ y + l_2\ y + l_2 + l_2 + l_0 + l_0 + 3. \tag{7}$$
   Simplifying (7) we get the constraint $s3.2.2.1$: $\{k_2 = 1, l_1 = l_2 = 0, l_0 = -3\}$. Simplifying (6) we get the constraint $s2.2.2.1$: $\{k_1 = 1, l_0 = -3, k_0 = 0, l_1 = 0\}$. Since these two rules have consistent constraints, they belong to a maximal consistent subset.

8. *Identifying Candidate lhs*: Abstracting the left sides of these two rules in MCS we get the left side of the conjecture $a(3, y)$.
9. *Compute* D_{lhs}: Using the function *compute* to determine a complete definition for $a(3, y)$ produces $D_{a(3,y)} = \{2.2.2.1, 3.2.2.1\}$.
10. *Find Right Side for* $a(3, y)$: Applying the most general solution $\delta = \{k_2 = 1, k_1 = 1, k_0 = 0, l_0 = -3, l_1 = 0, l_2 = 0\}$ to the conjecture $a(3, y) = 2^{k_1\ 3\ +\ k_2\ y + k_0} + l_1\ 3 + l_2\ y + l_0$ we get the lemma
$$6: \ a(3, y) = 2^{3+y} - 3,$$
which is added to the expanded ruleset and the process is repeated.
11. No additional lemmas are generated using the theory of $2exp$ since $a(4, y)$ and $a(x, 0)$ are both not expressible in $2exp$ and the Ackermann's function is monotonic in both its arguments.