Rewriting, Induction and Decision Procedures: A Case Study of Presburger Arithmetic

Deepak Kapur[*]

Department of Computer Science
University of New Mexico
Albuquerque, NM 87131
kapur@cs.unm.edu

1 Introduction

Theorem provers and automated reasoning tools are not as widely used in specification analysis, debugging and verification of hardware and software as one would hope. A major reason is perhaps that these tools are often found difficult to use. The learning curve for a typical theorem prover is quite high; it takes a great deal of effort by typical users to effectively use such a tool for their application domain. Even then, considerable resources must be expended by building a large knowledge base and library of useful properties in a representation suitable for the prover, to bring it to a level so that it can start playing a useful role.

For theorem provers to be acceptable to application experts, the key requirement is that a theorem prover should be able to easily perform reasoning steps considered routine in an application domain; performing such reasoning should definitely not become a burden on the expert. We have called this *reasoning in the large* and the inference steps used for such reasoning as *large* inference steps, to distinguish them from easy and application-independent inference steps typically supported in most proof checkers as well as many theorem provers, which we have called *reasoning in the small* [10]

In the computer-aided verification literature, there has recently been a surge of interest in decision procedures in order to enhance the capabilities of reasoning tools which operate in push-button mode. The main focus of these works has been mostly to fine-tune methods and frameworks for combining decision procedures, thus reviving earlier work by Shostak [22] and Nelson and Oppen [21], or integrating decision procedures with model-checkers and OBDD based tools [4]. Little attention has been paid to integrate decision procedures with methods for mechanizing induction, and/or with the use of the definitions of function symbols and their known properties in the form of lemmas.

We consider the following features crucial in building *large* inference steps:

1. Integrating decision procedures for equality, numbers, bits, bit vectors, finite sets, finite lists, and other frequently used data structures in applications,
2. performing case analyses automatically,
3. finding instances of definitions and lemmas using decision procedures for simplification, and
4. discharging conditions in definitions and lemmas.

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

Another approach for building large inference steps is by providing mechanisms for writing tactics which can be used to combine small inference steps to have the effect of a large inference step. Many proof checkers including PVS, ISABELLE, HOL, do provide linguistic constructs for writing tactics. Every user however must define such tactics, or use tactics from the library developed by other users. Further, it is unclear how large inference steps discussed below that can require intricate integration between decision procedures, simplification (rewriting) and induction, can be written as efficient tactics.

In this paper, we use the case of Presburger arithmetic for illustrating interaction between decision procedures, rewriting (simplification) and induction. The discussion is in the context of our rewrite-based theorem prover *Rewrite Rule Laboratory,* (*RRL*), which has been used to automatically verify properties of arithmetic hardware circuits including a family of adders, multipliers as well as the SRT division circuit [14, 15, 16, 19]. After briefly reviewing conditional rewriting, the cover set induction method, and Presburger arithmetic, we discuss how a decision procedure can be used to automatically find appropriate instantiations of rewrite rules serving as function definitions and lemmas, thus building large inference steps. Intertwining of simplification by rewriting and decision procedures is illustrated, thus exhibiting tight integration, something we conjecture, is nontrivial to achieve using tactics in tactic-based provers.

The next key issue is the interaction between decision procedures and induction, a topic that has not gotten much attention. We show how methods based on extracting induction schemes from function definitions can be generalized to exploit semantic information expressed as decision procedures. It is also shown how decision procedures can be augmented with induction schemes extracted from a family of function definitions so that a certain subclass of conjectures can be automatically decided, thus enlarging the scope of formulas that can be decided using traditional decision procedures. The proposed approach has the ability of making decision procedures extensible by enriching them with different induction schemes suggested by different terminating function definitions.

RRL supports decision procedures for (i) equality on ground terms (congruence closure), (ii) propositional reasoning, (iii) bits, (iv) freely generated data structures, as well as (v) a procedure for quantifier-free theory of Presburger arithmetic over integers and naturals [20, 12]. A distinguishing feature of these decision procedures is that if a formula cannot be found unsatisfiable, then equalities are generated as consequences to be used as rewrite rules for simplification. We believe that it is the integration of decision procedures with rewriting and induction which makes *RRL* effective for the application of reasoning about arithmetic circuits.

2 Background: Rewriting

Below, we give an overview of the rewrite-based approach to theorem proving and simplification.

Let $T(F, X)$ be the set of terms constructed using a finite set F of function symbols and a set X of variables. A term is either a variable $x \in X$ or $f(t_1, \ldots, t_n)$, where $f \in F$, and each $t_i, 1 \leq i \leq n$, is a term. 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, and they can be uniquely identified using *positions,* finite sequences of positive integer separated by "."'s. The subterm of t at the empty sequence ϵ is t itself. If $f(t_1, \cdots, t_n)$ is a subterm at position p, then t_j is the subterm at position $p.j$. A term $f(t_1, \cdots, t_n)$ 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 $\{x_1 \leftarrow t_1, \cdots, x_n \leftarrow t_n\}$, $n \geq 0$, and x_i's are distinct. θ applied on $s = f(s_1, \cdots, s_m)$, written as $\theta(s)$, 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 $\theta(s) = \theta(t)$.

An (unconditional) rewrite rule is $s \to t$ with $Vars(t) \subseteq Vars(s)$ and $s \notin X$. A rule $s \to t$ is *applicable* to a term u iff for some substitution θ and position p in u, $\theta(s) = u|_p$, the subterm at position p in u.

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 $\to_R$. $s \to_R t$ denotes rewriting of s to t by a single application of a rule in R. $\to_R^+$ and $\to_R^*$ denote the transitive and the reflexive, transitive closure of $\to_R$, respectively.

Conditional rewrite rules are interesting because of ways their conditions can be discharged. A conditional rewrite rule

$$s \to t \textit{ if } cond$$

is *applicable* to a term u iff for some substitution θ and position p in u, $\theta(s) = u|_p$ and $\theta(cond)$ can be reduced to $true$. Different definitions of conditional rewriting rely on different ways to ensure the truth of $\theta(cond)$ to check for the applicability of the conditional rule. In *RRL*, the congruence closure over ground terms, propositional calculus and Presburger arithmetic (see below) are used for discharging conditions. When u has a *context* in the form of a conjunction of literals, then the context can also be used to establish conditions; this is how contextual rewriting is defined; for details, see [24, 19].

3 Cover Set Induction: Cover Sets and Induction Schemes

The *cover set* method is used to implement well-founded induction in *RRL*, and has been used to successfully perform proofs by induction in a variety of nontrivial application domains including verification of arithmetic circuit descriptions [15, 14, 16]. Below, we give an overview of the cover set induction method. For details, the reader can refer to [25, 24, 23].

Function symbols in F are partitioned into *defined* and *interpreted* functions. An interpreted function symbol comes from a decidable theory $\mathcal{T}$ whose decision procedure is being integrated into rewriting and induction. Constructor symbols are viewed as interpreted symbols; the quantifier-free theory for free constructor symbols is decidable.

A defined function symbol f is specified by a finite set of terminating rewrite rules, and it is assumed that some rule in the definition covers every tuple of arguments to f (see below for a discussion on checking completeness of a definition). The termination of rewrite rules defining functions is established using well-founded rewrite orderings. *RRL*, for instance, uses the lexicographic recursive path ordering on terms induced by a well-founded precedence ordering on function symbols [5].

Given a complete function definition as a finite set of terminating rules

$$\{l_i \to r_i \textit{ if } cond_i \mid l_i = f(s_1, \cdots, s_n),\ 1 \leq i \leq k\},$$

the main steps of the cover set induction method are

1. *Generate a Cover Set from a Function Definition:* A cover set associated with f is a finite set of triples. For a rule $l \to r \; if \; cond$, where $l = f(s_1, \cdots, s_n)$, and $f(t_1^i, \cdots, t_n^i)$ is the i^{th} recursive call to f in r, the corresponding triple is $\langle\langle s_1, \cdots, s_n\rangle, \{\cdots, \langle t_1^i, \cdots, t_n^i\rangle, \cdots\}, \{cond\}\rangle$. The second component of a triple is the empty set if there is no recursive call to f in r. The third component is the empty set for unconditional rules. The positions in the recursive calls to f where the arguments change are called *inductive* (or *changeable*) positions. Other positions are called *unchangeable* or *noninductive* positions.
2. *Generate an Induction Scheme using a Cover Set:* Given a conjecture C with a basic subterm $u = f(x_1, \cdots, x_n)$, the cover set of f can be chosen for generating an induction scheme as follows.
   An induction scheme is a finite set of induction cases, each of the form $\langle\sigma_c, \{\theta_i\}, \{cond\}\rangle$ generated from a cover set triple $\langle\langle s_1, \cdots, s_n\rangle, \{\cdots, \langle t_1^i, \cdots, t_n^i\rangle, \cdots\}, \{cond\}\rangle$ as follows[1]:
   $\sigma_c = \{x_1 \leftarrow s_1, \cdots, x_n \leftarrow s_n\}$ and $\theta_i = \{x_1 \leftarrow t_1^i, \cdots, x_n \leftarrow t_n^i\}$.[2] The variables in the basic subterm u in inductive argument positions are then the *induction variables* of the conjecture C.
3. *Generate 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 in the corresponding triple of the induction scheme is the empty set.

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 proofs of conjectures.

4 Presburger Arithmetic

Throughout this paper, Presburger arithmetic stands for the quantifier-free theory of numbers (rationals, integers or natural numbers) with numeric variables, the arithmetic operations (successor(s), predecessor(p) and addition($+$)) and the arithmetic relations $(>, <, \neq, =, \geq, \leq)$.[3] It also goes under the name of *linear arithmetic* [3].

The pure subtheory includes the above symbols, but for verification purposes, we need to include uninterpreted symbols as well. By a *linear* term, we mean either a number, a variable, or a term of the form $f(t_1, \cdots, t_n)$ where f is an arithmetic operation and each t_i, $1 \leq i \leq n$, is a (not necessarily linear) term. If

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

[2] To generate an induction scheme, the subterm u in a conjecture C is unified 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 if u is a basic term $f(x_1, \cdots, x_n)$. It also works if only variables in the inductive positions of u are distinct. The method can be used to generate induction schemes from other nonbasic subterms in a conjecture as well.

[3] Presburger proved in 1929 that the full first-order theory of numbers with the above stated symbols is decidable [6]. In our work, we have used only a quantifier-free subtheory of it.

p is an arithmetic relation, $p(t_1, \cdots, t_n)$ denotes a linear atom. Linear terms and atoms are called *pure* if they do not involve any uninterpreted symbols. It is easy to see that every atom can be translated to $s_i = t_i$ or $s_i \leq t_i$.

In this paper, by a Presburger arithmetic decision procedure PA, we assume an algorithm specified in terms of input and output as follows.

- *Input:* A set of equations and inequalities $LC : \{s_i \; op \; t_i | 1 \leq i \leq n\}$, where $op \in \{=, \leq\}$, s_i's and t_i's are pure linear terms.
- *Output:* No, if LC cannot be solved. Otherwise, yes with $\langle \sigma, NLI \rangle$, a solved (simplified) form of LC (i.e., $\sigma \cup NLI \leq LC$ in a well-founded ordering $<$ on terms and literals), where $\sigma = \{x_j \leftarrow u_j \mid 1 \leq i \leq k\}$ satisfying the occur-check (i.e., x_j does not appear in any u_i's), and (possibly, a disjunction of conjunction of) a finite set of normalized inequalities NLI obtained after applying σ (again implying that no x_j appears in NLI). Further, any implicit equality derivable from LC is made explicit, i.e., any equality $s = t$ derivable from LC should be such that $\sigma(s) - \sigma(t)$ simplifies to 0. For instance, if LC includes $E \leq F - G$ and $F \leq G + E$, then $E - F + G = 0$ must be derivable from σ.
- *Completeness and Soundness:* The solution sets $Soln(LC) = Soln(\langle \sigma, NLI \rangle)$, i.e., any m-tuple $e = \langle e_1, \cdots, e_m \rangle$ of numbers is a solution to LC (for every $s_i = t_i \in LC$, $\theta(s_i) - \theta(t_i) = 0$ and for every $s_i \leq t_i \in LC$, $\theta(s_i) - \theta(t_i) \leq 0$, where $\theta = \{x_1 \leftarrow e_1, \ldots, x_m \leftarrow e_m\}$) iff e is a solution of σ and NLI.

For equations, a solved form can be obtained using an algorithm for solving linear Diophantine equations and conditions on variables. For inequalities, Fourier's algorithm can be used to check for unsatisfiability; a byproduct of this algorithm is the detection of implicit equalities. Details can be found in [12, 13].

5 Rewriting and Decision Procedures

Apart from deciding formulas in its theory, another important role a decision procedure can play is to determine whether a rewrite rule in a function definition or proved as a lemma can be applied to simplify a given conjecture with the ultimate goal of checking the validity of the conjecture. In order to do this, it is imperative that the decision procedure be tightly integrated with rewriting. Most theorem provers based on decision procedures including PVS, STEP, SVC, do not seem to perform such tight integration between rewriting/simplification and decision procedures.

In *RRL*, decision procedures are not only used for checking the validity of a goal and simplifying it to a normal form, but also for selecting which rewrite rule is applicable on the goal. Decision procedures for congruence closure and Presburger arithmetic interact with the rewriting/matching algorithm. A rewrite rule may often appear not to be applicable for simplifying a goal; but with the aid of decision procedures, it can indeed be used to show the validity of the goal. In contrast, this has to be done by the user manually in other decision procedures based theorem provers; see an example below. Capabilities such as these are perhaps the main reason why *RRL* has been successful in automatically proving properties of arithmetic circuits, especially the SRT division circuit [19] as well as in generating readable and compact proofs.

Below, we illustrate two aspects of cooperation of decision procedures. As the reader would notice, the decision procedures for equality on ground terms, propositional calculus, and Presburger arithmetic interact with each other for simplification and deducing additional equalities. Further, appropriate instantiations for the rules are found by establishing the conditions of the rules. What would take many human-guided steps in many provers and proof checkers can be done in a single step in RRL.[4]

Assume the following rewrite rules are already known (either as a part of definitions and/or lemmas known about function symbols appearing in the conjecture):

$$\begin{array}{l} 1.\ min(x,y) \rightarrow y \quad if \quad max(x,y) = x, \\ 2.\ f(x) \leq g(x) \rightarrow true \quad if \quad p(x). \end{array}$$

Suppose the following conjecture over the integers is attempted:

$$(p(x) \wedge (x \leq max(x,y)) \wedge (z \leq f(max(x,y))) \wedge (0 < min(x,y)) \wedge (max(x,y) \leq x))$$

$$\supset (z < g(x) + y).$$

A proof by contradiction is attempted. First, the conjecture is negated and Skolemized; so variables are replaced by Skolem constants. The following goal must be shown to be unsatisfiable.

$$p(A) \wedge (A \leq max(A,B)) \wedge (L \leq f(max(A,B))) \wedge (0 < min(A,B))$$

$$\wedge (max(A,B) \leq A)) \wedge \neg(L < g(A) + B).$$

Linear inequalities are first transformed by PA to the form $s_i \leq c_i$, where s_i is a linear term and c_i is a number.

$$p(A) \wedge (A - max(A,B) \leq 0) \wedge (L - f(max(A,B)) \leq 0) \wedge (-min(A,B) \leq -1)$$

$$\wedge (max(A,B) - A \leq 0)) \wedge (g(A) + B - L \leq 0).$$

Below we give a brief overview of the interaction among the three decision procedures (propositional satisfiability, ground congruence closure and Presburger arithmetic PA) for simplification, along with a sketchy detail of steps performed by each decision procedure. For more details, the reader should consult [12].

1. Simplify using the propositional calculus decision procedure and check for propositional unsatisfiability of literals.
2. Process equalities on ground terms using the congruence closure procedure. In RRL, this is done using a ground completion procedure by first turning equalities into terminating rewrite rules, and then, normalizing them [12, 9]. Trivial equalities such as $t = t$ are reduced to $true$.
3. Equalities are propagated to the rest of the formula. This is done by normalizing the rest of the literals in the goal using the rewrite rules in the ground canonical system obtained from the equalities.
   Repeat steps 1-3 until there is no change.
4. Process arithmetic constraints using PA by eliminating variables. If not found unsatisfiable, deduce all implicit equalities by obtaining a solved form for equalities. Go to step 2.
   Repeat all of the above steps until nothing changes.

[4] Boyer and Moore's prover [2] may also be able to prove this conjecture without any user guidance.

The above procedure terminates since an invocation of a decision procedure either has no effect, or produces an output smaller than its input in a well-founded order on literals. We illustrate the procedure on the example.

Propositional satisfiability and congruence closure procedures have no effect in the first iteration on the above example. From the linear inequalities, an implicit equality $max(A, B) = A$ is derived by PA. This equality is used by the congruence closure procedure as a rewrite rule to simplify the inequality set to:

$$L - f(A) \leq 0, \; -min(A, B) \leq -1, \; g(A) + B - L \leq 0.$$

The key inference step in PA is to pair up linear inequalities in which a variable appears positively as well as negatively, and eliminate the variable from the pair of inequalities. Given $a_1 X + q_1 \leq e_1$ and $-b_1 X + q_2 \leq e_2$, where $a_1, b_1 > 0$, deduce a new inequality $b_1 q_1 + a_1 q_2 \leq b_1 e_1 + a_1 e_2$. In the above example, the linear term L is eliminated:

$$-min(A, B) \leq -1, \; -f(A) + g(A) + B \leq 0.$$

If during elimination, an inequality of the form $0 \leq 0$ is deduced, all inequalities participating in the derivation are then equalities, and they are made explicit. If an inequality of the form $0 \leq -c$, where $c > 0$, is deduced, then unsatisfiability is detected. If on the other hand, a variable appears only positively (or negatively) in inequalities, then the formula is likely to be satisfiable except for unsatisfiability arising due to solutions of equations over integers (or natural numbers) being sought. More details can be found in [12].

For the above example, no new equality can be deduced by the decision procedures. Further, the formula cannot be shown to be unsatisfiable. The result of the interaction among the three decision procedures is:

$$max(A, B) = A, \; P(A), \; -min(A, B) \leq -1, \; -f(A) + g(A) + B \leq 0.$$

When a goal cannot be shown unsatisfiable and no new equalities can be deduced from the interaction of the decision procedures, it is then checked whether a rewrite rule from the data base of definitions and lemmas applies. Literals in the goal are analyzed for matching the rewrite rules, starting with the maximal literals (in the same well-founded ordering used to ensure termination of the rewrite rules in the data base) in the goal. If a match with the left side of some rule is found, then a proof of the condition of the rule, if any, under the match is attempted. If successful, the instance of the rewrite rule is added as a new constraint, and simplification by decision procedures is repeated. Since every literal in the condition of a rewrite rule is smaller than its left side, and the result of rewriting by a rewrite rule is smaller in the well-founded ordering, the termination of simplification is ensured.

Continuing with the above example, rule 1 is applicable on the literal $min(A, B)$ in the goal, using $\{x \leftarrow A, \; y \leftarrow B\}$ since the condition of rule 1 under this instantiation, $max(A, B) = A$, follows from the equalities in the rest of the goal. Literal $min(A, B)$ is reduced to B; $min(A, B) = B$ is added to the equality set. Simplification using the three decision procedures (after replacing $min(A, B) = A$ and eliminating B) then produces:

$$max(A, B) = A, \; P(A), \; -f(A) + g(A) \leq -1.$$

Assuming the ordering $f > g$, the next literal to consider in the subgoal is: $-f(A) + g(A) \leq -1$; the maximal subterm $f(A)$ in it matches a maximal term in the second linear rule since $P(A)$ is true. The instance of rule 2 is added to the inequality set, giving:

$$-f(A) + g(A) \leq -1,\ f(A) - g(A) \leq 0.$$

The decision procedure PA detects a contradiction from these inequalities, thus asserting the validity of the original conjecture.

This example is a good illustration of the tight integration of contextual rewriting with decision procedures. As the reader would notice, the original goal can be automatically established. The verification of the properties of the SRT division circuit includes many such proofs of formulas; details can be found in [16, 19, 17]. More details of the integration of contextual rewriting with decision procedures are given in [12].

6 Induction and Decision Procedures

We show how a decision procedure such as PA can be used to generalize the cover set method, to make it more effective for generating and analyzing induction schemes using unification modulo a decidable theory. Below, we provide a brief overview; more details can be found in [13].

Many data structures can be represented in many different ways; for example, numbers can be represented using 0 and s (to stand for the *successor* function) as constructors as well as in terms of $0, 1$ and $+$. Some functions, such as $gcd, divides, rem$, etc., are more convenient, and easier to define using $+$, instead of s. Finite sets can be represented using the *null* set, a constructor that *inserts* an element, or alternatively, using the null set, *singleton* sets and the *union* operation. Similarly, lists can be represented using the *empty* list and the *cons* constructor, or alternatively, the empty list, *singleton* lists and the *append* operator on lists. While reasoning about functions defined on such data structures, there is a need to convert from one representation to the other. This could be because (i) a function definition is given using a mixture of operators defined on different representations, (ii) a conjecture is expressed using functions which are defined on two different representations, or (iii) a conjecture involves a function that is defined using one representation but its arguments in the conjecture use a different representation.

Using PA, we illustrate how semantic information about a data structure can reconcile its different representations while attempting a proof of a conjecture. It thus becomes critical to generate an induction scheme using semantic information, that enables the application of induction hypotheses which otherwise cannot be applied because of different uses of a function.

We can generalize the algorithm for generating an induction scheme for a conjecture C by performing semantic unification, instead of syntactic unification, on a selected subterm $t = f(t_1, \cdots, t_n)$ in C, and the cover set associated with f. Given a cover set triple,

$$\mathrm{c} = \langle\langle s_1, \cdots, s_n\rangle, \{\cdots, \langle s_1^j, \cdots, s_n^j\rangle, \cdots\}, cond\rangle,$$

constraint equations $\{t_i = s_i \mid 1 \leq i \leq n\}$ with context $cond$ are solved; PA generates a substitution σ_c of $Vars(t) \cup Vars(\mathrm{c})$ (if the third component of c is empty then the context is assumed to be $true$). Similarly, substitutions θ_j of $Vars(t) \cup Vars(\mathrm{c})$ for induction hypotheses are generated using PA on the constraint equations $\{t_i = \sigma_c(s_i^j) \mid 1 \leq i \leq n\}$ with context $\sigma_c(cond)$.

As an example, consider the definition of `gcd` (greatest common divisor):

```
1. gcd(0, z)   --> z,                 2. gcd(z, 0)   --> z,
3. gcd(w, z+w) --> gcd(w, z) if w>0,  4. gcd(z+w, z) --> gcd(w,z) if z>0.
```

The cover set generated from `gcd` is:

```
Cover(gcd):{<<0,z>,{},{}>, <<z,0>,{},{}>, <<w, z+w>, {<w, z>}, {w>0}>,
            <<z+w, z>,{<w, z>},{z>0}>}}.
```

As the reader would have noticed, the induction scheme generated from this cover set is very different from the principle of mathematical induction. While proving a simple conjecture such as `gcd(u, v) = gcd(v, u)`, one would notice that the principle of mathematical induction is often not helpful, whereas the induction scheme proposed by the above cover set method immediately gives a proof, which shows the power of the cover set method.

For attempting a proof of the conjecture `gcd(u + u, s(s(0))) = s(s(0))`, however, the principle of mathematical induction suffices. This scheme can also be generated from the above cover set and the conjecture using the above algorithm:

```
{<{ u <- 0}, {}, {}>, <{ u <- 1}, {}, {}>, <{ u <- v + 1}, {{ u <- v}}, {}>}.
```

For instance, the first component of the first triple is the most general unifier (mgu) of $u + u = 0, s(s(0)) = z$, obtained from the first triple of the cover set. The second triple of the cover set does not yield any substitution since $0 = s(s(0))$ cannot be solved. The third triple of the cover set leads to $u + u = w, s(s(0)) = z + w, w > 0$ which gives the substitution for the conclusion in the second triple in the induction scheme; there is no induction hypothesis available for use since $u + u = s(s(0)), s(s(0)) = 0$ has no solution. The fourth triple in the cover set gives $u + u = z + w, s(s(0)) = z, z > 0$, giving the substitution $u \leftarrow v + 1$ for the conclusion; the induction hypothesis is generated from the substitution solving $u + u = v + v, s(s(0)) = s(s(0)), s(s(0)) > 0$.

6.1 Checking Completeness using a Decision Procedure

Before a function definition can be used to generate a sound and complete induction scheme using the cover set method, it must be proved terminating as well as complete. It must be the case that for any ground substitution σ_g of variables in $f(x_1, \dots, x_n)$, some rule $f(s_1^i, \dots, s_n^i) \rightarrow r_i$ *if* $cond_i$ from the definition of f must be applicable on $\sigma_g(f(x_1, \dots, x_n))$. Below we review a generalization of the methods discussed in [7, 11] so that definitions given using linear terms can also be checked for completeness using PA.

Checking for completeness amounts to constructing a formula that is a disjunction of conditions under which each rule covers the arguments of f. Assuming that $x_1, \dots, x_n$ are new variables not appearing in $s_1^i, \dots, s_n^i$, a function f is *completely defined* using rules

$$f(s_1^i, \dots, s_n^i) \rightarrow r_i \textit{ if } cond_i, \;\; 1 \le i \le k$$

if and only if the formula

$$\vee_i \;\; \exists y_1, \dots y_m, (cond_i \wedge (x_1 = s_1^i \wedge \dots x_n = s_n^i)),$$

is valid, where $y_1, \dots, y_m$ are free variables in $s_1^i, \dots, s_n^i$. If this formula is in a decidable theory, then its validity can be decided.

For instance, `gcd(u, v)` above is completely defined on natural numbers as shown below: We get

$$[\exists z(u = 0 \wedge v = z) \vee \exists z(u = z \wedge v = 0) \vee \exists z, w(w > 0 \wedge (u = w \wedge v = z + w))$$
$$\vee \exists z, w(z > 0 \wedge (u = z + w \wedge v = z))].$$

Using the decision procedure for Presburger arithmetic, the existential quantifiers can be eliminated, and the above formula simplifies to:

$$((u = 0) \vee (v = 0) \vee (v \geq u \wedge u > 0) \vee (u \geq v \wedge v > 0)),$$

a valid formula in Presburger arithmetic. The cover set generated from the definition of *gcd* is, thus, complete.

Methods discussed in [7, 11] cannot be used to prove the completeness of *gcd* since they depend upon the use of completion procedures; the completion procedure would not terminate on the definition of *gcd*.

If a formula generated for checking completeness is not valid, then the definition of f may or may not be complete depending upon whether the rewrite rules defining f are confluent or not. However, the associated cover set is incomplete. For more details, an interested reader may consult [8, 13].

If the arguments to f in the left and right sides of every rewrite rule in the definition of f are linear terms, and $cond_i$ is a conjunction of linear literals, then the formula corresponding to the completeness of the definition of f can be decided using a decision procedure for Presburger arithmetic.

The above method has the advantage that in case of an incomplete definition, it can list a finite representation of argument tuples on which the function f needs to be defined.

Incomplete Cover Sets In order to draw sound conclusions using the cover set method, it is important to use a complete cover set. Otherwise, unsound conclusions can be made. For example, if the cover set based on the following definition of *divides*:

```
1. divides(u, 0)        ->  true,
2. divides(u, v)        ->  false if (v < u) and not (v = 0),
3. divides(u, u + v)    ->  divides(u, v) if 0 < u.
```

is used without caring about its completeness, it is possible to wrongly conclude: $divides(x, y + y)\ \ if\ \ divides(x, y)$, even though the conjecture does not hold if $x = 0 \wedge y > 0$. The formula is true under the condition that $x > 0 \vee y = 0$, the values on which *divides* is defined. For $x = 0 \wedge y \neq 0$, $divides(x, y)$ is not defined; unless the above formula is viewed in a logic in which all undefined values are treated the same, the validity of the formula cannot be determined. The modified conjecture $divides(x, y + y)\ \ if\ \ divides(x, y) \wedge (x > 0 \vee y = 0)$ is a theorem provable by induction.

Using the induction scheme generated from the incomplete cover set of *divides*, $divides(2, x) = not(divides(2, s(x)))$ can be proved since *divides* is completely defined if its first argument is 2 (this also follows from the fact that the instantiation of the above condition, $(2 > 0 \vee x = 0)$, is true). This induction scheme can also be used to prove $divides(1, y)$. These examples suggest that an incomplete cover set can be used for proofs insofar as conjectures are relativized with respect to the condition characterizing the input subdomain on which the function definition is given; a detailed discussion can be found in [13].

6.2 Generalization Heuristic and Lemma Speculation

One of the key challenges in mechanizing proofs by induction is how to discover intermediate lemmas needed for a proof. Intermediate conjectures are generated during proof attempts. In some cases, however, it is possible to prove a generalization of a conjecture (from which the conjecture follows). Most induction theorem provers support heuristics for generalizing conjectures. Semantic analysis using a decision procedure can be used to improve upon the generalization heuristics.

An easy but useful heuristic is to abstract common subterms in a conjecture by a variable [20, 2]. Generalizations so obtained can be first quickly tried on a few test cases to make sure that they are at least true for some instantiations before a proof is attempted. If a particular generalization cannot be proved (or turns out to be not valid), another one can be tried after backtracking.

Given a conjecture C of the form, $l = r \textit{ if } cond$, RRL looks for a maximal nonvariable subterm s occurring in at least two of l, r and $cond$. In case of many such maximal subterms, many generalizations are possible. A generalized version C_g of C can be obtained by simultaneously abstracting (a subset of) distinct common subterms in C by distinct new variables. If C_g can be proved, then C is valid, whereas the converse is not necessarily true.

A weakness of the above heuristic is the requirement that term being generalized to a variable, have multiple syntactic occurrences. A subterm may not have multiple occurrences syntactically in a conjecture, even though semantically equivalent subterms may occur in it. Consider the following conjecture about *gcd*:

$$gcd(x + y + 1, 2 * x + 2 * y + 2) = x + y + 1.$$

If no semantic analysis is performed, a possible generalization by abstracting occurrences of $x + y + 1$ by a variable u is $gcd(u, 2 * x + 2 * y + 2) = u$. This generalization can be easily shown to be false.

From a semantic standpoint, however, the second argument of *gcd* is closely related to its first argument. Using PA for identifying equivalent subterms, this relationship can be identified, and the first argument of *gcd* can be found to appear twice in its second argument. The above conjecture can thus be generalized to: $gcd(u, u + u) = u$, which can be proved.[5]

The generalization heuristic can be modified to exploit semantic analysis as follows. Given a conjecture $l = r \textit{ if } cond$, find a maximal nonvariable subterm s of l that also appears in r and/or *cond*. This can be done by checking whether s appears syntactically, or whether for some other linear subterm t, $t \geq s$. If the answer is no, then s does not occur in t; otherwise, we find the number of times s appears in t (this can be done by repeated queries and subtraction from t until the result becomes smaller than s). Let $t = k*s+tr$, where k is a positive integer and tr is a linear term $< s$. Subterm s can be abstracted to be a new variable u, and t can be replaced by $k * u + tr$. A detailed description of the extended generalization procedure using PA is in [13].

7 Extending Induction Schemes with Decision Procedures

Most fully-automatic reasoning tools implementing OBDDs and model checkers have a major weakness, namely they are unable to perform any proof by induction; see [1]. As anyone attempting proofs by hand must have experienced, there

[5] Under the condition $u > 0$, the conjecture rewrites using the fourth rule of *gcd* to give $gcd(u, u) = u$, which can be proved. For the condition $u = 0$, the conjecture is again established by any of the rules 1 and 2 after simplification.

are many challenges in discovering a proof by induction—choosing variables to perform induction, induction schemes (e.g., whether to use the principle of mathematical induction or the principle of complete (Noetherian) induction in the case of natural numbers), and speculating intermediate lemmas needed in proofs.

Syntactic conditions can be developed which guarantee that certain equational conjectures whose proofs need the use of induction, can be decided automatically. By putting restrictions on conjectures and the definitions of function symbols appearing in conjectures, it can be ensured that using the cover set method, every subgoal in a proof attempt can be simplified to a formula in a decidable theory. We review this below. A detailed discussion can be found in [18]

Definition 7.1 *A definition of a function symbol f is $\mathcal{T}$-based in a decidable theory $\mathcal{T}$ if for each rule $f(t_1, \cdots, t_m) \to r$ in the definition, every t_i, $1 \le i \le m$, is an interpreted term in $\mathcal{T}$, any recursive call to f in r only has 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 $\mathcal{T}$.*

For examples, the definitions of `double, log, exp2`, and `*` below are $\mathcal{T}$-based over Presburger arithmetic; `double` is also $\mathcal{T}$-based over the theory of free constructors `0, s`.

```
 1. double(0)          --> 0,      2. double(s(x)) --> s(s(double(x))).
 3. log(0)             --> 0,      4. log(s(0))    --> 0,
 5. log(s(x)+s(x))     --> s(log(s(x))),
 6. log(s(s(x)+s(x))) --> s(log(s(x))).
 7. exp2(0)            --> s(0),   8. exp2(s(x))   --> exp2(x)+exp2(x).
 9. x * 0              --> 0,     10. x * s(y)     --> x + (x * y).
```

Definition 7.2 *A conjecture $f(x_1, \cdots, x_n) = r$ is called* **simple** *over $\mathcal{T}$ if f has a $\mathcal{T}$-based definition, and r is interpreted in $\mathcal{T}$.*

The conjecture `(C1)` below about `double` is simple over Presburger arithmetic, but `(C2)` about `log` is not.

```
(C1): double(m) = m + m,          (C2): log(exp2(m)) = m.
```

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. Further, formulas arising from induction subgoals can be simplified, and after the application of the induction hypothesis(es), the resulting formulas are in a decidable theory. Thus,

Theorem 1. *A simple conjecture C over $\mathcal{T}$ is decidable by the cover set method.*

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

To decide complex conjectures, the choice of induction schemes must be limited, and the interaction among the definitions of function symbols in the conjectures has to be analyzed. In [23], such an analysis is undertaken for predicting a priori failure of proof attempts when certain induction schemes are used. The notion of **compatibility** of function definitions is introduced for identifying intermediate steps in a proof which get blocked in the absence of additional lemmas. The same idea is used below to identify conditions under which the decidability of conjectures can be predicted a priori.

We first consider compatibility among the definitions of two function symbols. Later this is generalized to a sequence of function definitions.

Definition 7.3 *A $\mathcal{T}$-based term t is composed if*

1. *t is a basic term $f(x_1, \cdots, x_k)$, where f is $\mathcal{T}$-based, or*
2. *(a) $t = f(s_1, \cdots, t', \cdots, s_k)$, where t' is in an inductive position of a $\mathcal{T}$-based f and is composed, and each s_i is an interpreted term, and*
   (b) variables 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 may appear elsewhere in t.

The left side of `(C2)`, `log(exp2(m))`, is a composed term, for instance, in which `exp2(m)` is a basic subterm. The definition of a composed term guarantees that there is exactly one basic subterm, and that too in its innermost position.

Given a conjecture of the form $l = r$, where l is composed and r is interpreted, there is only one basic term in it whose outermost symbol is a defined symbol and $\mathcal{T}$-based. For a conjecture $f(t_1, \cdots, g(x_1, \cdots, x_k), \cdots, t_m) = r$, interaction between the right sides of rules defining g and the left side of rules defining f must be considered, as captured below in the property of *compatibility.*

Definition 7.4 *A definition of f is compatible with a definition of g in its i-th argument in $\mathcal{T}$ if for each right side r_g of a rule defining g:*

1. *if r_g is interpreted in $\mathcal{T}$, then $f(x_1, \cdots, r_g, \cdots, x_m)$ rewrites to an interpreted term in $\mathcal{T}$, and*
2. *if $r_g = h(s_1, \cdots, g(t_1, \cdots, t_k), \cdots, s_n)$ with 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 $\mathcal{T}$.*[6]

These requirements ensure that induction hypothesis(es) are applicable after function definitions have been applied. The above definition also applies to an interpreted function symbol (i.e., one in $\mathcal{T}$) and a $\mathcal{T}$-based function symbol. As an example, `+` in Presburger arithmetic is compatible with `*` with the usual recursive definition because of the associativity and commutativity properties of `+`, which are valid formulas of PA.

A conjecture in which a composed term is equated to an interpreted term, can be decided if all the symbols from the root to the position p of the basic subterm 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* $\langle f_1, \cdots, f_d \rangle$, of definitions of function symbols, where each f_i is compatible with f_{i+1} at j_i-th argument, $1 \le i \le d-1$. The above ideas are best illustrated using the following example:

```
(C3): bton(pad0(ntob(m))) = m.
```

Functions `bton` and `ntob` convert binary representations to numbers, and vice versa, respectively. The function `pad0` adds a leading binary zero to a bit vector. These functions are commonly used to reason about number-theoretic properties of parameterized arithmetic circuits [15, 14]. 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. The above equation captures an important property that the padding does not affect the number

[6] A more general requirement can be given in case of multiple recursive calls.

output by a circuit. The underlying theory is the combination of the quantifier-free theory of Presburger arithmetic, the quantifier-free theory of bit vectors with the quantifier-free theory of free constructors `nil`, `cons`, and `b0`, `b1`, to stand for binary 0 and 1. (In the definitions below, bits increase in significance in a bit vector (list) with its first element being the least significant.) Function symbols `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 there is no way to rewrite `ntob(x + x)` and `ntob(s(x + x))`).

A proof attempt of `(C3)` leads to two basis and two step subgoals based on the cover set of `ntob`. In the first basis subgoal where `m <- 0`, `bton(pad0(ntob(0))) = 0`, `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, giving the formula `0 + 0 + 0 + 0 = 0`, which is valid 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)`. Subterm `ntob(s(s(x2 + x2)))` in 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))))` which on the hypothesis application, gives 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 similarly to a valid formula in Presburger arithmetic. Hence, the validity of `(C3)` 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.

Theorem 2. *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 $\langle f_1, f_2, \cdots, f_{d-1}, f_d \rangle$ of function symbols from the outermost function symbol f_1 of l to the basic subterm $f_d(x_1, \cdots, x_m)$ is compatible.*

Acknowledgement: Most of this work has been done jointly with my former student Mahadevan Subramaniam. I thank Jürgen Giesl for helpful comments.

References

1. A. Gupta, *Inductive Boolean Function Manipulation: A Hardware Verification Methodology for Automatic Induction.* Ph.D. Thesis, CMU, Pittsburgh, 1994.

2. R.S. Boyer and J S. Moore, *A Computational Logic Handbook.* Academic Press, 1988.
3. R.S. Boyer and J S. Moore, "Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic", *Machine Intelligence* 11, P. Hayes, D.Mitchie and J. Richards (eds), 1988.
4. R.E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. on Computers*, C-35(8), 1986.
5. N. Dershowitz, "Termination of rewriting," *J. Symbolic Computation,* 3, 69-116, 1986.
6. H. Enderton, *A Mathematical Introduction to Logic.* Academic Press.
7. J.-P. Jouannaud and E. Kounalis, "Automatic proofs by induction in theories without constructors," *Information and Computation* 82, 1-33, 1989.
8. 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.
9. D. Kapur, "Shostak's congruence closure as completion," Proc. *Intl. Conf. on Rewriting Techniques and Applications* (*RTA-97*), Barcelona, Spain, June 1997.
10. D. Kapur, "Rewriting, decision procedures and lemma speculation for automated hardware verification," Proc. *Theorem Provers in Higher-order Logics,* (ed. Gunter and Felty), Springer LNCS 1275, Murray Hill, 1997, 171-182.
11. D. Kapur, P. Narendran, D. Rosenkrantz, and H. Zhang, "Sufficient-completeness, quasi-reducibility and their complexity," *Acta Informatica,* 28, 1991, 311-350.
12. D. Kapur and X. Nie, "Reasoning about numbers in Tecton," Proc. *8th Intl. Symp. Methodologies for Intelligent Systems*, Charlotte, North Carolina, 1994, 57-70.
13. D. Kapur and M. Subramaniam, "New uses of linear arithmetic in automated theorem proving for induction," *J. Automated Reasoning,* 16(1-2), 1996, 39-78.
14. D. Kapur and M. Subramaniam, "Mechanically verifying a family of multiplier circuits," Proc. *CAV,* (eds. Alur and Henzinger), LNCS 1102, 1996, 135-146.
15. D. Kapur and M. Subramaniam, "Mechanical verification of adder circuits using powerlists," *J. Formal Methods in System Design,* 13(2), Sep. 1998, 127-158.
16. D. Kapur and M. Subramaniam "Mechanizing reasoning about arithmetic circuits: SRT division," Proc. *17th FSTTCS,* Springer LNCS 1346, Kharagpur,India, 1997.
17. D. Kapur and M. Subramaniam, "Mechanizing reasoning about large finite tables in a rewrite based theorem prover,", Proc. *ASIAN-98,* LNCS 1538, Manila, 1998.
18. D. Kapur and M. Subramaniam, "Extending decision procedures with induction schemes" Proc. *CADE-17* (ed. McAllester) Springer LNAI 1831, Pittsburgh, 2000.
19. D. Kapur and M. Subramaniam, "Using an induction prover for verifying arithmetic circuits," *J. Software Tools for Technology Transfer,* Springer Verlag, Sep 2000.
20. D. Kapur and H. Zhang, "An overview of Rewrite Rule Laboratory (RRL)," *J. of Computer and Mathematics with Applications,* 29, 2, 1995, 91-114.
21. G. Nelson and D.C. Oppen, "Simplification by cooperating decision procedures," *ACM Trans. on Programming Languages and Systems* 1 (2), 1979, 245-257.
22. R.E. Shostak, "Deciding combination of theories," *J. ACM* 31 (1), 1984, 1-12.
23. M. Subramaniam, *Failure Analyses in Inductive Theorem Provers.* Ph.D. Thesis, Department of Computer Science, University of Albany, New York, 1997.
24. H. Zhang, "Implementing contextual rewriting," Proc. *3rd Intl. Workshop on Conditional Term Rewriting Systems,* (eds. Remy and Rusinowitch), Springer LNCS 656, France, 1992, 363-377.
25. H. Zhang, D. Kapur, and M.S. Krishnamoorthy, "A mechanizable induction principle for equational specifications," Proc. *9th Intl. Conf. Automated Deduction (CADE),* (eds. Lusk and Overbeek), Springer LNCS 310, Chicago 1988, 250-265.