Automatic Synthesis of Decision Procedures: a Case Study of Ground and Linear Arithmetic*

Predrag Janičić[1] and Alan Bundy[2]

[1] Faculty of Mathematics, University of Belgrade
Studentski trg 16, 11000 Belgrade, Serbia — email: janicic@matf.bg.ac.yu
[2] School of Informatics, University of Edinburgh
Appleton Tower, Crichton St, Edinburgh EH8 9LE, UK — email: A.Bundy@ed.ac.uk

Abstract. We address the problem of automatic synthesis of decision procedures. Our synthesis mechanism consists of several stages and submechanisms and is well-suited to the proof-planning paradigm. The system (ADEPTUS), that we present in this paper, synthesised a decision procedure for ground arithmetic completely automatically and it used some specific method generators in generating a decision procedure for linear arithmetic, in only a few seconds of CPU time. We believe that this approach can lead to automated assistance in constructing decision procedures and to more reliable implementations of decision procedures.

1 Introduction

Decision procedures are often vital in theorem proving [2, 7]. In order to have decision procedures usable in a theorem prover, it is necessary to have them implemented not only efficiently, but also flexibly. It is often very important to have decision procedures for new, user-defined theories. Also, the implementation of decision procedures should be such that it can be verified in some formal way. For all these reasons, it would be fruitful if the process (or, at least, all its routine steps) of implementing decision procedures can be automated. It would help in avoiding human mistakes in implementing decision procedures.

In this paper we follow ideas from the programme on proof plans for normalisations and for automatic generation of decision procedures from [4]. As discussed there, many steps of many decision procedures can be described via sets of rewrite rules (so, object level proofs could also be relatively easily derived). Following and extending the ideas from [4], we have developed a system ADEPTUS (coming from *Assembly of DEcision Procedures via TransmUtation and Synthesis*) capable of automatically synthesising normalisation procedures and decision procedures.[1] All the methods that ADEPTUS generates are built in

* First author supported by EPSRC grant GR/R52954/01 and Serbian Ministry of Science grant 144030. Second author supported in part by EPSRC grant GR/S01771.

[1] Adeptus (Lat.) is also "one with the alchemical knowledge to turn base metals into gold". ADEPTUS is implemented in PROLOG as a stand-alone system. The code and the longer version of this paper are available from www.matf.bg.ac.yu/~janicic.

the spirit of the proof planning paradigm (and are implemented in PROLOG). For some theories, the approach gives not only automatically generated decision procedures, but also — by generating structured procedures consisting of simple methods — a higher-level understanding of syntactical transformations within the theory. Also, thanks to their modular architecture, generated procedures can be easily modified to slightly changed circumstances. We believe that this approach can be helpful in providing an easier and more reliable implementation of decision procedures. In this paper we evaluate our techniques on ground arithmetic and linear arithmetic (over rationals). ADEPTUS synthesised the decision procedures for ground arithmetic in around 3 seconds, and a decision procedure for (quantified) linear arithmetic in around 5 seconds of CPU time.

2 Preliminaries

Decision procedure. A theory $\mathcal{T}$ is decidable if there is an algorithm, which we call a *decision procedure*, such that for an input $\mathcal{T}$-sentence f, it returns *yes* if and only if $\mathcal{T} \vdash f$ (i.e., if f is a theorem of $\mathcal{T}$), and returns *no* otherwise).

Ground and linear arithmetic. Ground arithmetic is a fragment of arithmetic that does not involve variables. Linear arithmetic is a fragment of arithmetic that involves only addition (nx is treated as $x + \cdots + x$, where x appears n times). For both these theories, we assume that variables can range over rational numbers. The Fourier/Motzkin procedure [9] is one of the decision procedures for linear arithmetic.

Backus-Naur form. For describing syntactical classes, we use Backus-Naur form — BNF (equivalent to context-free grammars). We assume that each BNF specification has attached its *top nonterminal.* The language of a BNF is a set of all expressions that can be derived from the top nonterminal. For representing some infinite syntactical classes, for convenience, we use some meta-level conditions. We define the relation *ec* (*element of class*) as follows: $ec(b, e, c)$ holds iff e can be derived from c w.r.t. the BNF specification b.

Rewrite rules. Unconditional rewrite rules are of the form: $RuleName : l \longrightarrow r$. Conditional rewrite rules are of the form: $RuleName : l \longrightarrow r \; if \; p_1, p_2, \ldots, p_n$, where p_1, p_2, …, p_n are literals. These rewrite rules may be used modulo the underlying theory $\mathcal{T}$ (e.g., the rule $n_1x + n_2x \longrightarrow nx \;\; if \;\; n = n_1 + n_2$ may be used modulo linear arithmetic). For a rule $RuleName : \;\; l \longrightarrow r \; if \; p_1, p_2, \ldots, p_n$, we say that it is *sound* w.r.t. $\mathcal{T}$ if for arbitrary $\mathcal{T}$-formula Φ and arbitrary substitution φ it holds that $\mathcal{T} \vdash \Phi$ if $\mathcal{T}$, $p_1\varphi$, $p_2\varphi$, …, $p_n\varphi \vdash \Phi[l\varphi \mapsto r\varphi]$, and we say that it is *complete* w.r.t. $\mathcal{T}$ if for arbitrary $\mathcal{T}$-formula and arbitrary substitution φ it holds that $\mathcal{T} \vdash \Phi$ only if $\mathcal{T}, p_1\varphi, p_2\varphi, \ldots, p_n\varphi \vdash \Phi[l\varphi \mapsto r\varphi]$.[2]

Proof planning and methods. Proof-planning is a technique for guiding the search for a proof in automated theorem proving. To prove a conjecture, within a proof-planning system, a method constructs the proof plan and this plan

[2] Note that $\mathcal{T}$ does not necessarily contain the theory of equality, so we define soundness and completeness of the rules this way.

is then used to guide the construction of the proof itself [3]. These plans are made up of tactics, which represent common patterns of reasoning. A method is a specification of a tactic. A method has several slots: a name, input, preconditions, transformation, output, postconditions, and the name of the attached tactic. A method cannot be applied if its preconditions are not met. Also, with the transformation performed and the output computed, the postconditions are checked and the method application fails if they fail.[3]

3 Proposed Programme

Our programme (slightly modified from the first version [4]) for automated synthesis of normalisation methods and decision procedures has several parts:

- Given a syntactical class, a set of rewrite rules, and a kind of transformation, select (if it is possible) a subset of rewrite rules that is sufficient to transform any member of the input syntactical class in the required way. The output class should also be generated automatically. We call a *method generator* an algorithm capable of generating a method that transforms members of the input class to members of the output class.
- There are different kinds of methods, e.g., one for removing some function symbol, one for stratification, one for thinning etc. (see further text and [4] for explanation of these terms); for each of them, there is a method generator.
- Given several generated methods, it should be possible to combine them (automatically) into a compound method or, sometimes, into a decision procedure for some theory;
- Methods (and compound methods) should be designed in such a way that their soundness, completeness, and termination can be easily proved;
- Since some transformations (required for some procedures) are very complex, building methods may require human interaction and assistance.

Example 1. From any formula derivable from f w.r.t. the following BNF:

$f ::= af|\neg f|f \vee f|f \wedge f|f \Rightarrow f|f \Leftrightarrow f|(\exists var : sort)f|(\forall var : sort)f$

(where af is another nonterminal, describing atomic formulae) the symbol $\Leftrightarrow$ can be removed by exhaustively using the rewrite rule $f_1 \Leftrightarrow f_2 \longrightarrow (f_1 \Rightarrow f_2) \wedge (f_2 \Rightarrow f_1)$ and the resulting formula can be derived from f w.r.t. the following BNF:

$f ::= af|\neg f|f \vee f|f \wedge f|f \Rightarrow f|(\exists var : sort)f|(\forall var : sort)f$.

Following the above programme, we implemented our system ADEPTUS capable of generating code for real-world decision procedures. We have implemented several method generators. They take a given BNF, transform it into another one, and build a method that can transform any formula that belongs to the first BNF into a formula that belongs to the second BNF. On the set of all these generators, we can perform a (heuristically guided) search for a sequence of methods which goes from a starting BNF to a trivial BNF (consisting of only $\top$ and $\bot$). If the final syntactical class is equal to $\{\bot, \top\}$, then the whole of the sequence yields

[3] Alteratively, instead of (active) postconditions, methods can have (passive) effects — conditions that are *guaranteed* to be true when the method succeeds.

a decision procedure for the underlying theory (under some assumptions about available rewrite rules). If such a method can be built, soundness, termination, and completeness can be easily proved. Apart from these method generators, we also use special-purpose method generators. For simplicity, in the rest of the paper we assume that, in formulae being transformed, variables are standardised apart, i.e., there are no two quantifiers with the same variable symbol.

4 Method Generators and Generated Methods

Normalisation Method Generators. Normalisation methods are methods based on exhaustive application of rewrite rules. Each normalisation method has the following general form:

name: *methodname*;
input: f;
preconditions: $ec(b, f, \mathit{top\ nonterminal})$ (where b is the input BNF);
transformation: transforms f to f' by exhaustive application of the set of rewrite rules (applying to positions that correspond to the attached language constructs);
output: f';
postconditions: $ec(b', f', \mathit{top\ nonterminal})$ (where b' is the output BNF).

We have implemented generators for several kinds of methods:

Remove is a normalisation method used to eliminate a certain function symbol, predicate symbol, logical connective, or a quantifier from a formula. The method uses sets of appropriate rewrite rules and applies them exhaustively to the current formula until no occurrences of the specific symbol remain. For instance, as shown in Example 1, the given BNF specification can be transformed to the corresponding BNF specification without the symbol $\Leftrightarrow$.

Stratify is a normalisation method used to stratify one syntactical class into two syntactical classes containing some predicate or function symbols, logical connectives or quantifiers. For instance, a stratify method for moving disjunctions beneath conjunctions can be constructed if the following rewrite rules are available: `st_conj_disj1`: $f_1 \wedge (f_2 \vee f_3) \longrightarrow (f_1 \wedge f_2) \vee (f_1 \wedge f_3)$, `st_conj_disj2`: $(f_2 \vee f_3) \wedge f_1 \longrightarrow (f_2 \wedge f_1) \vee (f_3 \wedge f_1)$.

Thin is a normalisation method that eliminates multiple occurrences of a unary logical connective or a unary function symbol. For instance, we can use the rule $\neg\neg f \longrightarrow f$ in order to transform each formula derivable from $f ::= af|\neg f$ to a formula derivable from $f ::= af|\neg af$.

Absorb is a normalisation method that can eliminate some recursion rules. For instance, we can use the rule `rm_mult`: $c_1 \cdot c_2 \longrightarrow c_3 \quad \text{if} \quad c_3 = c_1 \cdot c_2$ in order to transform each term derivable from $t ::= t \cdot rc|rc$ (where the nonterminal rc denotes rational constants) to a term derivable from $t ::= rc$.

Left-assoc is one of the normalisation methods for reorganising within a class. If a syntactical class contains only one function symbol or a connective and if that symbol is both binary and associative, then members of this class can be put into left associative form. For instance, we will need the left association of addition and the left association of conjunction.

A *normalisation method generator* is a procedure with the following input: *(i)* a BNF b for the input expressions; *(ii)* a set of rewrite rules R; *(iii)* a kind of the required method (e.g., *remove*). It generates a method $\mathcal{M}$ and a BNF b' (for the output expressions).[4] By applying the rules from R, $\mathcal{M}$ transforms any expression derivable from b to an expression derivable from b'.

Example 2. Consider the BNF: $f ::= h(a)|h(b)|g_1(a)|g_2(b)$ where a and b are nonterminals, and the following rewrite rules: $R_1 : h(x) \longrightarrow g_1(x)$, $R_2 : h(x) \longrightarrow g_2(x)$. These rules are sufficient for eliminating the symbol h and for transforming the above BNF into: $f ::= g_1(a)|g_2(b)$. However, it cannot be reached by arbitrary use of exhaustive applications of the given rewrite rules: R_1 should be applied only to $h(a)$, and R_2 only to $h(b)$. The lesson is that we have to take care about which rule we use for specific language constructs. This sort of information has to be built into the method we want to construct.

A *normalisation method generator* works, basically, as follows: first it tries to eliminate non-recursive nonterminals in the input BNF, then searches for "problematic" BNF rules and generates the output BNF set, then, a generic algorithm for searching over available rewrite rules is invoked and it checks if all "problematic" language constructs can be rewritten in such a way that any input formula, when rewritten, is derivable from the top nonterminal of the output BNF. Also, this search mechanism attaches rewrite rules to particular language constructs. If there are no required rewrite rules, a method generator reports it, so the user could try to provide missing rules (in a planned, advanced version, which is not part of the work presented in this paper, the method generator would speculate the remaining necessary rules and/or try to redefine/relax the output class).

Example 3. The remove method generator can generate the method for removing the symbol $\neg$ from formula derivable from f w.r.t. the following BNF:

$$f ::= f \vee f|f \wedge f|\neg\bot|\neg\top|\neg t < t|\neg t = t|\bot|\top|t < t|t = t|$$

with the following rewrite rules attached to particular language constructs:

`rm_bottom`:	$\neg\bot \longrightarrow \top$	attached to $\neg\bot$
`rm_top`:,	$\neg\top \longrightarrow \bot$	attached to $\neg\top$
`rm_neg_less`:	$\neg(t_1 < t_2) \longrightarrow (t_2 < t_1) \vee (t_1 = t_2)$	attached to $\neg t < t$
`rm_neg_eq`:	$\neg(t_1 = t_2) \longrightarrow (t_1 < t_2) \vee (t_2 < t_1)$	attached to $\neg t = t$

The output BNF is: $f ::= f \vee f|f \wedge f|\bot|\top|t < t|t = t|$ and, by the generated remove method, the formula $\neg(3 < 2) \wedge \neg(1 = 2)$ will be transformed to $(2 < 3 \vee 3 = 2) \wedge (1 < 2 \vee 2 < 1)$.

Special-Purpose Method Generators. The first one of the following special-purpose generators can be used for a quantifier elimination procedure for any theory, while the remaining three are specific for linear arithmetic. Note, however, that it is essential to have these generators (although they are theory-specific): they can be used in an automatic search process and generate the required methods with the given preconditions (which are not known in advance).

[4] In our system, the tactics are not implemented yet. So, our procedures produce meta-level proof plans, not the object level proofs.

Method Generator for Adjusting the Innermost Quantifier. It generates a method that transforms a formula in prenex normal form in the following way: if its innermost quantifier is existential, then keep it unchanged; if its innermost quantifier is universal, then rewrite the formula $(Qx_1)(Qx_2)\dots(Qx_n)(\forall x)f$ to $(Qx_1)(Qx_2)\dots(Qx_n)\neg(\exists x)\neg f$ by using the following rewrite rule: `rm_univ`: $(\forall x)f \longrightarrow \neg(\exists x)\neg f$. The motive of this method is to deal only with elimination of existential quantifiers.

One-side Method Generator. It generates a method that transforms all literals in such a way that each of them has 0 as its second argument. For instance, for symbols $<$, $>$, $\leq$, $\neq$, $\geq$, $=$ as parameters, after applying the generated *one-side* method, each literal will have one of the following forms: $t < 0$, $t > 0$, $t \leq 0$, $t \neq 0$, $t \geq 0$, $t = 0$.

Method Generator for Isolating a Variable. It generates a method that isolates a distinguished variable x in all literals. After applying that method, each of the literals either does not involve x or has one of the forms: $\alpha x = \beta$, $x = \beta$, $\alpha x < \beta$, $x < \beta$ (where α and β have no occurrences of x).

Method Generator for Removing a Variable. The *cross multiply and add* step is the essential step of the Fourier/Motzkin's procedure [9]. It is applied for elimination of x from $\exists x F(x)$, where F is in disjunctive normal form and each of its literals either does not involve x or has one of the forms: $\alpha x = \beta$, $x = \beta$, $\alpha x < \beta$, $x < \beta$ (where α and β have no occurrences of x). After performing this step, x does not occur in the current formula and so the corresponding quantifier can be deleted. It is important to have this generator (instead of a single method) — it generates required methods with concrete specific preconditions and postconditions, which is vital for combining with other concrete methods, and for automatic search process.

Properties of Generated Methods. A normalisation method links two sets of formulae. From the syntactical point of view, each formula f_1 derivable from the top nonterminal of the input BNF should be transformed (in a finite number of steps) into a formula f_2 derivable from the top nonterminal of the output BNF. From the deductive point of view, it should hold that $\mathcal{T} \vdash f_1$ if (and only if) $\mathcal{T} \vdash f_2$. If the "if" condition holds, then the method is sound, and if the "only if" condition holds then the method is complete (w.r.t. $\mathcal{T}$).

Termination. For each generated method it must be shown that it is terminating (by considering properties of the rewrite rules used[5]). For some sorts of methods, their termination is guaranteed by the way they are generated.

Soundness. We distinguish soundness of a method w.r.t. syntactical restrictions and soundness of a method w.r.t. the underlying theory $\mathcal{T}$:

- If a method transforms one formula into another one, then it is ensured by the method's postconditions that the second one does meet the re-

[5] Note that these sets of rewrite rules are not always confluent. Moreover, for certain tasks, such as, for instance, transforming a formula into disjunctive normal form, there is no confluent and terminating rewrite system [10].

quired syntactical restrictions (given by the method specification), so the method is sound w.r.t. syntactical restrictions.[6]
  - All available rewrite rules (all of them correspond to the underlying theory $\mathcal{T}$) are assumed to be sound. Thus, since a method is (usually) based on exhaustive application of some (normally sound) rewrite rules, it is trivially sound w.r.t. $\mathcal{T}$.

Completeness. We distinguish completeness of a method w.r.t. syntactical restrictions and w.r.t. the underlying theory $\mathcal{T}$:

  - It is not *a priori* guaranteed that a method can transform any input formula (which meets the preconditions) into some other formula (that belongs to the output class), i.e., it is not guaranteed that the method is complete w.r.t. syntactical restrictions. Namely, a method maybe uses some conditional rewrite rules (which cannot be applied to all input formulae). If a method uses only unconditional rewrite rules or conditional rewrite rules which cover all possible cases, then it can transform any input formula into a formula belonging to the output class.
  - Completeness of a method w.r.t. $\mathcal{T}$ relies on the completeness of the rewrite rules used. If a method can transform any input formula into a formula belonging to the output class and if all the rewrite rules it uses are complete, then the method is complete w.r.t. $\mathcal{T}$.

5 Search Engine for Synthesising Compound Methods

Given method generators, a BNF description of a theory $\mathcal{T}$, and a set of available rewrite rules, a user can try to combine different generated methods and transform the initial BNF step by step, searching for some goal BNF. Also, an automatic search for compound methods or a decision procedure for $\mathcal{T}$ can be performed. The goal of this process is to generate a sequence of methods such that: *(i)* the output BNF of a non-last method is the input BNF of the next method in the sequence; *(ii)* the output BNF of the last method in the sequence is a goal BNF, for instance, a trivial BNF — consisting only of rules with $\top$ and $\bot$ for the top nonterminal. Of course, this sequence can have more methods that are different instances of the same kind of methods, or even the very same method more than once. In each step, our search procedure tries all available method generators, with all possible parameters (based on the underlying language). In order to ensure termination, the search procedure tries to find a sequence of methods

[6] Conditional rules are the reason for using active postconditions in methods (instead of passive *effects*). For instance, for BNF $f ::= f \wedge f | n = n | n < n | \top | \bot$, the rewrite rules **rm_ls1**: $n_1 < n_2 \longrightarrow \top$ if $number(n_1)$, $number(n_2)$, $n_1 < n_2$ and **rm_ls2**: $n_1 < n_2 \longrightarrow \bot$ if $number(n_1)$, $number(n_2)$, $n_1 > n_2$ eliminate the symbol $<$. The method generator would take both these rules for building a remove method for $<$, but (since it works only in syntactical manner) it would not check if the conditions for **rm_ls1** and **rm_ls2** cover all cases, i.e., if the generated method can transform *any* input formula. That is why the methods have (active) postconditions that check if the input formula is really rewritten so the result belongs to the output class.

that consists of subsequences, such that each of them is of length less than or equal to a fixed value M, and such that the last BNFs of the subsequences are of strictly decreasing size. So, in any generated procedure there might be some BNF size increasing steps (for instance, with introducing new symbols in the current BNF), but the whole of the generated procedure will be size decreasing. The size of BNF specification is a heuristic measure and we define it to be the sum of sizes of all its rules; the size of a rule $c ::= c'$ is equal to $100 \cdot n_c(c') + 10 \cdot n_1(c') + n_2(c')$, where n_c denotes the number of occurrences of c in c', n_1 the number of occurrences of all other nonterminals in c', and n_2 the number of all other symbols in c'. Defined this way, the measure forces the engine to try to get rid of recursive nonterminals and then of the nonterminals whose specifications involve some other nonterminals. The trivial, goal BNF (consisting of only $f ::= \top|\bot$) has the size 2. If the current sequence cannot be continued, the engine backtracks and tries to find alternatives.

Example 4. The size of the following BNF $f ::= af|\neg f, \quad af ::= \top|\bot$ is 113 (10 for $f ::= af$, 1+100 for $f ::= \neg f$, 1 for $af ::= \top$, 1 for $af ::= \bot$).

The size of the following BNF specification (for ground arithmetic):

$$\begin{aligned} f &::= af|\neg f|f \vee f|f \wedge f|f \Rightarrow f|f \Leftrightarrow f \\ af &::= \top|\bot|t = t|t < t|t > t|t \leq t|t \geq t|t \neq t \\ t &::= rc|-t|t \cdot t|t + t \end{aligned}$$

is 1556 (af denotes atomic formulae, t denotes terms, and rc denotes rational constants). The size of BNF for the full linear arithmetic is 2233.

Given a finite number of method generators and a finite number of rewrite rules, at each step a finite number of methods can be generated (there is also a finite number of possible parameters). Thus, since the algorithm produces subsequences (of maximal length M) of decreasing sizes (that are natural numbers) of corresponding BNF specifications, the given algorithm terminates. If method generators can generate all methods necessary for building the required compound method, then (thanks to backtracking) the given algorithm can build one such compound method (for M large enough). If we iterate the given algorithm (for $M = 1, 2, 3, \ldots$), then it will eventually build the required compound method, so this iterated algorithm is complete. However, we can also use it only with particular values for M (then the procedure is not complete, but it gives better results if it used only for an appropriate value for M).

The ordering of method generators is not relevant for termination and correctness of the search algorithm, but it is important for its efficiency. We used the following ordering (based on empirical tests, simpler than a potential theoretical analysis, specific for each case): `remove`, `thin`, `absorb`, `stratify`, `left_assoc`.

When normalisation methods themselves cannot build a decision procedure, we use special-purpose method generators and the basic search engine in a more complex way. The search for a decision procedure based on quantifier elimination is performed in three stages, by the following *compound search engine*:

- the first stage is reaching a BNF for which the method for adjusting the innermost quantifier is applicable;

- the second stage produces a sequence of methods (that will form a loop) for variable elimination; the output BNF of this sequence of methods has to be a subset of its input BNF;
- the third stage is for final simplifications, it starts with the output BNF of the first stage, but with all rules involving variables and quantifiers deleted; its goal BNF specification is the trivial one (i.e., it consists only of $\top$ and $\bot$).

For each of these stages we use the basic search engine and we use all method generators with higher priority given to the special-purpose method generators.

Properties of Compound Methods. A set of generated methods for some underlying theory $\mathcal{T}$ can be combined (by a human, or automatically) into a compound method (for that theory). Compound methods (in this context) can use primitive methods in a sequence or in a loop (but not conditional branching). The preconditions of a compound method are the preconditions of the first method, and the postconditions are the postconditions of the last method used.[7]

Termination. If a compound method is a sequence of terminating methods, then it is (trivially) terminating. If it has a loop, a deeper argument is required.

Soundness. Since it relies on the soundness of the used primitive methods, every compound method is also sound (both w.r.t. syntactical restrictions and w.r.t. the underlying theory $\mathcal{T}$). Meeting the syntactical restrictions of the compound method is also ensured by its postconditions.

Completeness. If all the used methods are complete and if the compound method is terminating, then it is (trivially) complete. More precisely, if a compound method (*i*) is terminating; (*ii*) uses only (primitive) methods which never fail (i.e., the methods which transform any input formula to a formula belonging to the output class) and which use only complete rewrite rules, then that compound method is complete (w.r.t. $\mathcal{T}$).

Based on the above considerations, we can make a crucial observation: if a compound method for some theory $\mathcal{T}$ has an input BNF corresponding to the whole of $\mathcal{T}$, a trivial output BNF consisting only of $\top$ and $\bot$, and if it is terminating, sound, and complete (w.r.t. $\mathcal{T}$)[8], then it is a decision procedure for $\mathcal{T}$. This way, we can, in some cases, trivially get a proof that some (automatically generated) compound method is a decision procedure for some theory.

6 Evaluation

We ran the basic search engine, on the BNF specification for ground arithmetic given in Example 4, with $M = 3$, with the described method generators, and with 59 relevant rewrite rules available. We set the goal BNF specification to be

[7] This way of constructing the preconditions and postconditions of a compound method is not adequate in general but suffices for the examples we were working on (recall that in compound methods that our system generates, the output BNF of a method is always the input BNF of the next method in the sequence).

[8] Soundness and completeness properties rely on properties of the rewrite rules used.

the trivial one ($f ::= \top|\bot$), thus aiming at synthesising a decision procedures for ground arithmetic. The search algorithm took 2.91 seconds of *cpu* time[9], during the search there were 48 methods successfully generated and there are 22 of them in the final sequence. The search algorithm produced the sequence of methods `DP_GA` with the following "overview" (in bracket the sizes of the output BNFs are given): *remove* $\Leftrightarrow$ *(1345), remove* $\Rightarrow$ *(1144), remove* $\leq$ *(1123), remove* $\geq$ *(1102), remove* $\neq$ *(1081), remove* $>$ *(1060), remove* $-$ *(959), stratify* $[\wedge, \vee]$ *(969), thin* $\neg$ *(906), remove* $\neg$ *(858), stratify* $[\vee]$ *(868), stratify* $[+]$ *(878), left_assoc* $\vee$ *(788), left_assoc* $+$ *(698), left_assoc* $*$ *(608), absorb* $*$ *(487), absorb* $+$ *(366), remove* $<$ *(345), remove* $=$ *(324), left_assoc* $\wedge$ *(327), remove* $\wedge$ *(206), remove* $\vee$ *(2).*

Example 5. The method stratify [+] from the above list was generated for the following input BNF:

$$f ::= f_1|f \vee f$$
$$f_1 ::= f_1 \wedge f_1|\bot|\top|t < t|t = t|$$
$$t ::= t \cdot t|t + t|rc$$

with the following rewrite rules attached to particular language constructs (derivable from t):

`st_mult_plus1`: $t_1 \cdot (t_2 + t_3) \longrightarrow (t_1 \cdot t_2) + (t_1 \cdot t_3)$ attached to $t \cdot (t + t)$
`st_mult_plus2`: $(t_2 + t_3) \cdot t_1 \longrightarrow (t_2 \cdot t_1) + (t_3 \cdot t_1)$ attached to $(t + t) \cdot t$

The output BNF is:

$$f ::= f_1|f \vee f$$
$$f_1 ::= f_1 \wedge f_1|\bot|\top|t < t|t = t|$$
$$t ::= t_1|t + t$$
$$t_1 ::= t_1 \cdot t_1|rc$$

By this method, the formula $2\cdot(1+3) < 3$ will be transformed to $2\cdot1+2\cdot3 < 3$.

Theorem 1. *The procedure* `DP_GA` *for ground arithmetic is terminating, sound and complete, i.e., it is a decision procedure for ground arithmetic.*

Proof sketch. The procedure `DP_GA` is sound and terminating, as all generated methods are sound and terminating and there is no loop. We still don't claim that it is complete as there are some conditional rewrite rules used. For instance, in the step *absorb* $+$ of `DP_GA`, the conditional rule `reduce_plus`: $t_1 + t_2 \Rightarrow t_3$, if $t_3 = t_1 + t_2$ is used, but it is still not shown that its condition covers all possible cases. The user can show this by proving: $(\forall c_1 : rational)(\forall c_2 : rational)(\exists c_3 : rational)(c_3 = c_1 + c_2)$. It is easy to prove that such conjectures are theorems of arithmetic. Moreover, some of them can be proved by the decision procedure `DP_LA` for linear arithmetic (which we also automatically generated, see the subsequent text). All this leads us to conclude that the procedure `DP_GA` is correct.

Example 6. This example shows the formulae produced by the 22 subsequent methods of the procedure `DP_GA` applied to the formula $\neg(7 \leq 5) \Rightarrow \neg(2\cdot(1+3) \geq 3)$ (it is assumed that $\wedge$ has higher priority than $\vee$).

[9] The system is implemented in SWI Prolog and tested on a 512Mb PC Celeron 2.4Ghz.

remove $\Leftrightarrow$	$\neg(7 \leq 5) \Rightarrow \neg(2\cdot(1+3) \geq 3)$
remove $\Rightarrow$	$\neg(\neg(7 \leq 5)) \vee \neg(2\cdot(1+3) \geq 3)$
remove $\leq$	$\neg(\neg(7 < 5 \vee 7 = 5)) \vee \neg(2\cdot(1+3) \geq 3)$
remove $\geq$	$\neg(\neg(7 < 5 \vee 7 = 5)) \vee \neg(3 < 2\cdot(1+3) \vee 2\cdot(1+3) = 3)$
remove $\neq$	$\neg(\neg(7 < 5 \vee 7 = 5)) \vee \neg(3 < 2\cdot(1+3) \vee 2\cdot(1+3) = 3)$
remove $>$	$\neg(\neg(7 < 5 \vee 7 = 5)) \vee \neg(3 < 2\cdot(1+3) \vee 2\cdot(1+3) = 3)$
remove $-,$	$\neg(\neg(7 < 5 \vee 7 = 5)) \vee \neg(3 < 2\cdot(1+3) \vee 2\cdot(1+3) = 3)$
stratify $[\wedge, \vee]$	$(\neg(\neg 7 < 5) \vee \neg(\neg 7 = 5)) \vee \neg 3 < 2\cdot(1+3) \wedge \neg 2\cdot(1+3) = 3$
thin $\neg$	$(7 < 5 \vee 7 = 5) \vee \neg 3 < 2\cdot(1+3) \wedge \neg 2\cdot(1+3) = 3$
remove $\neg$	$(7 < 5 \vee 7 = 5) \vee (2\cdot(1+3) < 3 \vee 3 = 2\cdot(1+3)) \wedge 2\cdot(1+3) < 3 \vee 3 < 2\cdot(1+3)$
stratify $[\vee]$	$(7 < 5 \vee 7 = 5) \vee ((2\cdot(1+3) < 3 \wedge 2\cdot(1+3) < 3) \vee 3 = 2\cdot(1+3) \wedge 2\cdot(1+3) < 3) \vee$ $(2\cdot(1+3) < 3 \wedge 3 < 2\cdot(1+3)) \vee 3 = 2\cdot(1+3) \wedge 3 < 2\cdot(1+3)$
stratify $[+]$	$(7 < 5 \vee 7 = 5) \vee ((2\cdot 1 + 2\cdot 3 < 3 \wedge 2\cdot 1 + 2\cdot 3 < 3) \vee 3 = 2\cdot 1 + 2\cdot 3 \wedge 2\cdot 1 + 2\cdot 3 < 3) \vee$ $(2\cdot 1 + 2\cdot 3 < 3 \wedge 3 < 2\cdot 1 + 2\cdot 3) \vee 3 = 2\cdot 1 + 2\cdot 3 \wedge 3 < 2\cdot 1 + 2\cdot 3$
left_assoc $\vee$	$((((7 < 5 \vee 7 = 5) \vee 2\cdot 1 + 2\cdot 3 < 3 \wedge 2\cdot 1 + 2\cdot 3 < 3) \vee 3 = 2\cdot 1 + 2\cdot 3 \wedge 2\cdot 1 + 2\cdot 3 < 3) \vee$ $2\cdot 1 + 2\cdot 3 < 3 \wedge 3 < 2\cdot 1 + 2\cdot 3) \vee 3 = 2\cdot 1 + 2\cdot 3 \wedge 3 < 2\cdot 1 + 2\cdot 3$
left_assoc $+$	$((((7 < 5 \vee 7 = 5) \vee 2\cdot 1 + 2\cdot 3 < 3 \wedge 2\cdot 1 + 2\cdot 3 < 3) \vee 3 = 2\cdot 1 + 2\cdot 3 \wedge 2\cdot 1 + 2\cdot 3 < 3) \vee$ $2\cdot 1 + 2\cdot 3 < 3 \wedge 3 < 2\cdot 1 + 2\cdot 3) \vee 3 = 2\cdot 1 + 2\cdot 3 \wedge 3 < 2\cdot 1 + 2\cdot 3$
left_assoc $\cdot$	$((((7 < 5 \vee 7 = 5) \vee 2\cdot 1 + 2\cdot 3 < 3 \wedge 2\cdot 1 + 2\cdot 3 < 3) \vee 3 = 2\cdot 1 + 2\cdot 3 \wedge 2\cdot 1 + 2\cdot 3 < 3) \vee$ $2\cdot 1 + 2\cdot 3 < 3 \wedge 3 < 2\cdot 1 + 2\cdot 3) \vee 3 = 2\cdot 1 + 2\cdot 3 \wedge 3 < 2\cdot 1 + 2\cdot 3$
absorb $\cdot$	$((((7 < 5 \vee 7 = 5) \vee 2 + 6 < 3 \wedge 2 + 6 < 3) \vee 3 = 2 + 6 \wedge 2 + 6 < 3) \vee$ $2 + 6 < 3 \wedge 3 < 2 + 6) \vee 3 = 2 + 6 \wedge 3 < 2 + 6$
absorb $+$	$((((7 < 5 \vee 7 = 5) \vee 8 < 3 \wedge 8 < 3) \vee 3 = 8 \wedge 8 < 3) \vee 8 < 3 \wedge 3 < 8) \vee 3 = 8 \wedge 3 < 8$
remove $<$	$((((\bot \vee 7 = 5) \vee \bot \wedge \bot) \vee 3 = 8 \wedge \bot) \vee \bot \wedge \top) \vee 3 = 8 \wedge \top$
remove $=$	$((((\bot \vee \bot) \vee \bot \wedge \bot) \vee \bot \wedge \bot) \vee \bot \wedge \top) \vee \bot \wedge \top$
left_assoc $\wedge$	$((((\bot \vee \bot) \vee \bot \wedge \bot) \vee \bot \wedge \bot) \vee \bot \wedge \top) \vee \bot \wedge \top$
remove $\wedge$	$((((\bot \vee \bot) \vee \bot) \vee \bot) \vee \bot) \vee \bot$
remove $\vee$	$\bot$

We applied the compound search engine on the BNF description of the full linear arithmetic, with $M=3$ for the first and the third stage, with $M=5$ for the second stage[10], with all the described method generators, and with 71 relevant rewrite rules available. The search algorithm took 4.80 seconds of *cpu* time and during the search there were 89 methods successfully generated, while there are 51 of them in the final sequence, yielding a decision procedure `DP_LA` with:[11]

- 9 methods in the first stage: *remove* $\Leftrightarrow$, *remove* $\Rightarrow$, *remove* $\leq$, *remove* $\geq$, *remove* $\neq$, *remove* $>$, *remove -*, *remove -*, *stratify* $[\forall, \exists]$,
- 22 methods in the quantifier elimination loop: *adjust_innermost* x_0, *stratify* $[\wedge, \vee]$, *thin* $\neg$, *remove* $\neg$, *one_side* $[0, [<, >, \leq, \neq, \geq, =]]$, *stratify* $[\vee]$, *stratify* $[+]$, *stratify* $[+]$, *left_assoc* $\vee$, *left_assoc* $\wedge$, *stratify* $[<, =]$, *remove* $\wedge$, *left_assoc* $+$, *left_assoc* $+$, *left_assoc* $\cdot$, *absorb* $\cdot$, *absorb* $+$, *absorb* $\cdot$, *absorb* $+$, *isolate* $[[x_0, rc\cdot x_0], [<, >, \leq, \neq, \geq, =]]$, *eliminate* $[[x_0, rc\cdot x_0], [<, >, \leq, \neq, \geq, =]]$,
- 20 methods for final simplifications: *stratify* $[\wedge, \vee]$, *thin* $\neg$, *remove* $\neg$, *stratify* $[\vee]$, *stratify* $[+]$, *left_assoc* $\vee$, *stratify* $[+]$, *left_assoc* $+$, *left_assoc* $+$, *left_assoc* $\cdot$, *absorb* $\cdot$ *absorb* $+$, *absorb* $\cdot$, *absorb* $+$, *remove* $<$, *remove* $=$, *left_assoc* $\wedge$, *remove* $\wedge$, *remove* $\vee$, *remove* $\neg$

Theorem 2. *The procedure* `DP_LA` *for linear arithmetic is terminating, sound and complete, i.e., it is a decision procedure for linear arithmetic.*

Proof sketch. Each of individual methods used in the generated procedure `DP_LA` is terminating. Since each loop eliminates one variable and since there are a finite number of variables in the input formula, the loop terminates. Hence,

[10] For lower values of M the system failed to generate the required procedure.

[11] Same methods (e.g., left_assoc $+$) are applied to different language constructs.

the procedure DP_LA is terminating. Since all methods in DP_LA use only sound rewrite rules, all of them are sound, and hence, the procedure is sound. The completeness relies not only on the completeness of the rewrite rules used, but also on the coverage property for the methods that use conditional rewrite rules. It can be shown (similarly as for DP_GA) that all required coverage properties are fulfilled (moreover, some of the coverage properties can be proved by the generated procedure itself, which is acceptable, as we know that the procedure is sound). Therefore, in each method, either unconditional rules are used or conditional rules that cover all possible cases. Hence, all methods always succeed and are complete, and the procedure DP_LA is complete. All in all, the procedure DP_LA terminates, it transforms an arbitrary input (linear arithmetic) formula Φ into $\top$ or $\bot$, while the output is $\top$ iff Φ is a theorem of linear arithmetic.

We don't claim that the generated procedure DP_LA is the shortest or the most efficient one. However, we doubt that a decision procedure for linear arithmetic can be described in a much shorter way (see, for instance, the description from [5]). This suggests that it is non-trivial for a human programmer to implement this procedure without flaws and bugs, even when provided with the code for the key step (*cross multiply and add*), because the most probable flaws are rather in correctly combining all the remaining steps.

7 Related Work

Our approach is based on ideas from [4] and apart from that strong link, as we are aware of, it can be considered basically original.

The work presented here is related to the Knuth-Bendix completion procedure [8] and its variants in a sense that it performs automatic construction of decision procedures. However, there are significant differences. While the completion procedure generates a confluent and terminating set of rewrite rules, and hence a way how to reach a normal form, it does not give a description of the normal form. In contrast, our system does not necessarily produce a decision procedure (or a normalisation procedure) whenever the completion procedure, but when it does, it also provides a finite description of the output (normalised) language. The completion procedure generates procedures that are based on exhaustive applications of rewrite rules, while our system produces procedures that use subsets of rewrite rules in stages and give structured proofs (easily understandable to a human). For instance, our system can generate a procedure for constructing conjunctive normal form, which cannot be done by the completion procedure and by a single rule set (because, as said, there is no confluent and terminating rewrite system for transforming a formula into disjunctive normal form [10]). We believe that it would be worthwhile to combine our work with the Knuth-Bendix completion procedure in the following way: the completion procedure can be used to find a confluent and terminating set of rules and then ADEPTUS can be used over them.

Our work is also related to work aimed at deriving decision procedures using superposition-based inference system for clausal equational logic [1]. That approach is an alternative to the congruence closure algorithm and to the Knuth-Bendix completion procedure. It does not use subsets of rewrite rules in stages, and it cannot handle some transformations required for decision procedures for fragments of arithmetic.

The presented approach is also related to work that performs automatic learning of proof methods [6]. The system LEARNΩMATIC learns proof methods (including decision procedures) from proof traces obtained by brute force application of available primitive methods. This approach (unlike ours) does not give opportunities for simple proofs of termination or completeness of learnt methods.

8 Realm of the Approach and Further Automation

In the presented method generators, we take a method kind, input BNF, and a set of rewrite rules, and use them to generate a required method (with some output BNF). However, it would be fruitful if we could start with an input BNF and look at BNFs and methods that can be obtained by subsets of the available rewrite rules. It is interesting to consider if, for a given BNF and a set of (terminating) rewrite rules, we can compute the output BNF. The answer for the general case is negative, since the resulting set of expressions is not necessarily definable by a BNF. Even if there is an algorithm that (given a BNF and a terminating set of rewrite rules) constructs an output BNF *whenever it is possible* (this is subject of our current research[12]), it would still not ensure further automation of our programme in general case. Namely, if we want to synthesise a decision procedure, we would generate a sequence of BNFs looking for a trivial one and we would have to check if two BNFs give the same language, but that problem is undecidable. Therefore, it is likely that we cannot have a complete such procedure for synthesising decision procedures. On the other hand, we believe that the presented system can work well in many practical situations. It is heuristic and its realm is determined by the set of method generators available (so it is difficult to make a formal characterisation of the realm). Basically, it can be used for producing linear procedures, possibly with loops, but with no branching. In addition to linear arithmetic, it can be also used for producing decision procedures for other fragments of arithmetic (e.g., Presburger arithmetic) or for some normalisation procedures for some inductively defined data structures. Procedures for fragments of arithmetic are the most illustrative examples for the approach that we have found so far. We are looking for additional such illustrative theories.

The problem of combining decision procedures is not addressed by our approach: a decision procedure for a combination theory can be synthesised only if it as a whole can be described in terms of normalisation methods.

For future work we are planning the following lines of research: we will be looking for other challenging domains (for instance, it would be interesting to

[12] The algorithms for some special cases of this problem were presented by the authors and Alan Smaill at the workshops CIAO 2003, CIAO 2004, and at *Deduction and Applications* meeting at Dagstuhl, 2005., without publications. The work described in this paper has not been presented or published before.

use our system in the context of SMT (satisfiability modulo theory) solving, for producing modules for checking unsatisfiability for underlying theory); we will try to extend the set of our method generators and search engines and will try to further improve their efficiency; we will implement generators not only for methods, but also for the corresponding tactics; we will try to automate the process of checking if conditions in the rewrite rules used cover all possible cases (we will try to do it whenever possible by using the "self-reflection" principle, as discussed in the proofs of theorems 1 and 2); we will try to combine our system with Knuth-Bendix completion procedure.

9 Conclusions

We presented a system (ADEPTUS) for synthesising decision procedures, based on ideas from [4]. ADEPTUS consists of several method generators and mechanisms for searching over them and combining them. We have implemented the system and used it for automatically generating decision procedures (in PROLOG) for ground arithmetic and for linear arithmetic. These implementations are correct (and the system makes easier proving correctness, completeness and termination), which is not quite easy for a human programmer to achieve. The approach generates procedures that are structured and easy to understand, and also very modular, making it easy to adapt them to slightly changed circumstances (e.g., with new rules or terms introduced). We believe that our approach can be used in other domains as well and can lead to automation of some routine steps in different types of programming tasks.

References

1. Armando, A., S. Ranise, and M. Rusinowitch: 'Uniform Derivation of Decision Procedures by Superposition'. *CSL 2001*, Vol. 2142 of *LNCS*, Springer, 2001.
2. Boyer, R. S. and J. S. Moore: 'Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study of Linear Arithmetic'. *Machine Intelligence 11*. 1988,
3. Bundy, A.: 'The Use of Explicit Plans to Guide Inductive Proofs'. In: R. Lusk and R. Overbeek (eds.): *9th Conference on Automated Deduction*. 1988.
4. Bundy, A.: 'The Use of Proof Plans for Normalization'. In: R. S. Boyer (ed.): *Essays in Honor of Woody Bledsoe*, 1991.
5. Hodes, L.: 'Solving Problems by Formula Manipulation in Logic and Linear Inequalities'. In ProcIJCAI-71. 1971.
6. Jamnik, M., M. Kerber, M. Pollet, and C. Benzmuller: 'Automatic Learning of Proof Methods in Proof Planning'. CSRP-02-5, University of Birmingham, 2002.
7. Janičić, P. and A. Bundy: 'A General Setting for the Flexible Combining and Augmenting Decision Procedures'. *Journal of Automated Reasoning* **28**(3), 2002.
8. Knuth, D. E. and P. B. Bendix: 'Simple word problems in universal algebra'. In: J. Leech (ed.): *Computational problems in abstract algebra*. Pergamon Press, 1970.
9. Lassez, J.-L. and M. Maher: 'On Fourier's algorithm for linear arithmetic constraints'. *Journal of Automated Reasoning* **9**, 373–379, 1992.
10. Socher-Ambosius, R.: 'Boolean algebra admits no convergent rewriting system'. *4th Conference on Rewriting Techniques and Applications*, Vol. 488 of *LNCS*, 1991.